TURING 图灵程序设计丛书

New Riders

ActionScript for Multiplayer Games and Virtual Worlds

Learn multi-user interaction concepts from the experts

ActionScript 大型网页游戏开发

[美] Jobe Makar 著

李鑫 马舜 译





人民邮电出版社
POSTS & TELECOM PRESS



Jobe Makar

Electrotank公司的创办人、游戏及虚拟世界首席架构师，该公司主要为在线多人游戏和虚拟世界开发提供技术支持和服务。他是应用广泛的EUP虚拟世界和MMOG平台（www.eupsite.com）的开发人员之一。最近十年里，他共开发了200多款Flash游戏和9个虚拟世界。他还写过几本关于高级Flash、ActionScript和游戏编程的书。



李鑫

资深CG动画讲师及插画师，深度关切游戏与拉布拉多，喜欢研究小逻辑，最近空闲时在制作原创动画《瑜的门票》。相信交互式媒体的发展如一切美好事物般会在不经意间抹上眉梢，沿路风景比站台重要。



马舜

网名NickyMa，因喜欢后街男孩Nick而取此名。2003年末开始接触Flash，毕业后从事网站后端开发，后因喜欢Flash而转为前端，AS3开发经验丰富，目前为一款MMORPG做前端开发。从业经历涉及移动增值、金融股票、游戏开发等领域。闲暇时喜欢写开源软件，以分享技术为快乐，相信“坦诚于心，快乐就会无处不在”。



图灵程序设计丛书

ActionScript for Multiplayer Games and Virtual Worlds
Learn multi-user interaction concepts from the experts

ActionScript大型网页游戏开发

[美] Jobe Makar 著

李 鑫 马 舜 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

ActionScript大型网页游戏开发 / (美) 梅克(Makar,J.) 著；李鑫，马舜译. --北京：人民邮电出版社，2011.1
（图灵程序设计丛书）
ISBN 978-7-115-24271-6

Ⅰ. ①A… Ⅱ. ①梅… ②李… ③马… Ⅲ. ①动画－设计－图形软件，Flash ActionScript Ⅳ. ①TP391.41

中国版本图书馆CIP数据核字(2010)第221522号

内 容 提 要

本书是一本讲述用 ActionScript 3 进行大型网页游戏开发的教程。本书首先概述了网页游戏的特点及发展现状，然后分章介绍了聊天、逻辑决策、实时移动、大厅系统、等距视图、化身、用户之家等网页游戏的设计要点，最后详细介绍了实时坦克游戏。本书图文并茂，由简入繁，循序渐进，配以大量实例和游戏代码，适合学习借鉴。

本书适合从事网页游戏开发的中高级人员阅读参考。

图灵程序设计丛书
ActionScript大型网页游戏开发

◆ 著　[美] Jobe Makar
译　李 鑫 马 舜
责任编辑　杨海玲
执行编辑　罗 婧 罗词亮
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：15
字数：355千字　2011年1月第1版
印数：1－3 000册　2011年1月北京第1次印刷
著作权合同登记号　图字：01-2010-4748号
ISBN 978-7-115-24271-6

定价：45.00元
读者服务热线：(010)51095186　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

译 者 序

2009 年 10 月中旬，我刚刚帮好友 Psychedelico 译完了 PV3d 文档，机缘偶现，N 神的 Riabook 网站开始组群翻译本书，随即襄助。

在 Flash 及 Flex 的各种富媒体应用中，多人游戏与虚拟世界无疑是最容易被网民所接受乃至追捧的研发方向之一。或依附社交网站与之共赢，或独立自由门户广招拥趸；或精致隽永，或热辣快意……在 Flash Player 惊人的市场占有率之下，可供游戏开发者选用的商业模式及项目策略层出不穷。另外，由于体积上的优势，用 Flash 及 Flex 构建的多人网页游戏或微客户端虚拟世界开始逐步蚕食传统“重甲”网游的领地。目前，除却 3D 大型多人在线游戏开发之外，Flash 已经能够实现传统网游所能达到的全部内容及要求。

乘此煦阳东风暖，当赴万里瀚海游。若有此书作伴，你的开发路程必会顺畅不少。Jobe Makar 所领军的 Electrotank 是业内为数不多的几个较早开发 Flash 多人在线内容的公司之一，十余年的不懈努力使其成为该领域技术的顶级提供商。作为多年来技术的一次总结，该书不仅有着详实的理论及丰富的例证，更可贵的是，它还囊括了业内所用的很多高效技术：Socket 服务器、实时运动、区块式解算、A* 算法、等距视图、精灵序列图、化身系统……而且在公司论坛中，其各项目组的主程也在为各位读者及开发者耐心地答疑解惑，其对技术的袒露程度使人不禁觉得其公司名字没准儿起错了？（不像是那种封闭的重装机甲——兴许他们是更信奉“原力”的杰迪。）

也正是由于以上原因，本书原版面世伊始就引起了国内众多 Flash 游戏开发者的热议。作为译者，我们承受的压力可想而知。图灵风尚素来严谨、技术微末困惑异常、词句斟酌抓狂倒悬，以及种种未知的生活黑子爆发……这不是翻译，这是范 • 迪塞尔式的战斗！所以，为了这场完美偏执的盛宴，我们还需感谢下列人士。

N 神——没有来自你断续的激赏与鞭策，人可能就会不时地黯淡一把。另外问一句：Riabook 为何总能在第一时间找到那些好书呢？

李松峰——那叠批复得密密麻麻的初审稿确有醍醐之效。严格就是最好的兴奋剂。多谢您不吝训启！

罗婧——狮子座！琳琅背后孰料披星戴月，玲珑目下方显金石遗风，付梓之幸，当由汝成！

罗词亮——单人独骑平断这堆叠罗列的最后编审要务，莫非你是火法帝？

Slomka、圆号手、蜗牛の qipy、JING、★精灵☆ /kuk、Ryan Liu——绵绵默默，无私的帮助与鼓励；没有你们大声地喊“吁”，本书的问世可能会迟很多。

金钩高挂，且纳珠帘锦绣。但凡所有鉴析更误，皆可去图灵论坛、www.riabook.cn 处讨论或者直接去 Electrotank 的论坛提问。

最后，谨祝所有人平安好运！

李 鑫

2010 年 10 月

前　　言

在你正式开始本书学习之前，请先阅读以下几个方面的内容。这里，将向你介绍一下本书中将会用到的工具、本书范例源代码存放的位置，以及怎样使用这些源代码。如果你需要了解更多信息 , 可以随时通过电子邮件（jobe@electrotank.com）与作者联系。

本书主要讨论多人游戏的概念，以及如何使用 ActionScript 将其实施到项目中。我们假定你已经熟悉 ActionScript 和 Flash 的基础知识。

ActionScript 3 和 Flash Develop

本书中讨论的以及在范例文件中出现的所有客户端代码都是用 ActionScript 3 编写的，它们是基于 Adobe Flash Player 10 播放器的。学习本书并不需要了解 Adobe Flash CS4 或它的更新版本。所有的项目都是使用 Flex 编译器进行编译的。所有范例中的项目文件都是使用 Flash Develop 创建的（Flash Develop 是一个开源的使用 Flex 编译器的开发工具，可以在 www.flashdevelop.org 下载）。安装了 Flash Develop，你就可以轻松地打开和编译所有范例中的项目。如果你使用的是 Flex Builder，则需要把文件导入 Flex Builder 项目中。

ElectroServer服务器

本书使用 ElectroServer 4.06（www.electro-server.com）作为 Socket 服务器来为所有的多人项目范例提供服务器端支持。ElectroServer 是目前比较流行的为 Flash、Shockwave、Unity 多人游戏以及虚拟世界提供支持的 Socket 服务器之一。我们将在第 3 章中正式介绍它，并且全书都将使用它作为服务器。

范例文件

和本书有关的所有范例和游戏文件都放在压缩包 gamebook.zip 中，你可以从下面两个地址下载。

- 我的网站 http://www.electrotank.com/gamebook。
- 出版公司网站 http://www.peachpit.com/actionscriptformpgs。



注意　如果你使用 http://www.peachpit.com/actionscriptformpgs 这个地址，则需要注册并登录，然后输入本书的 ISBN，之后方能看到附件的下载地址。

在以上任一地址中 , 下载 gamebook.zip 文件并解压缩。然后可以看到各个章节的子文件夹。

范例的目录结构

我们为本书准备了很多范例。其中包括一个很大的名为“古老家园”的虚拟世界范例。下载了 gamebook.zip 文件后，就可以在其中的 old_world 文件夹下找到这个范例的源文件。其他范例的客户端代码都可以在相应章的文件夹中找到。本书在讨论到具体范例时还会提及它们的位置。

大多数范例也用到了服务器端代码。除了“古老家园”以外，所有客户端范例的服务器端代码都在 book_files/examples_extension 文件夹下。“古老家园”的所有代码在 book_files/old_world/server_extension 文件夹下。安装和运行这些文件的详细说明参见附录。

幕 后 英 雄

如果没有下面这些人的共同努力，本书是不可能完成的。



Mike Grundvig

Mike 编写了第 3 章。除此之外，他还为“古老家园”编写了服务器端代码并配置了数据库。本书中有好几章都会用到“古老家园”这个范例。

Teresa Carrigan

Teresa 为第 9 章的坦克游戏编写了全部的服务器端代码，同时也为全书中除了“古老家园”和合作游戏之外的每一个范例编写了服务器端代码。另外，Teresa 还为本书写了很多小节，这些小节讨论了服务器端代码和应用程序的部署。



Mike Bowen

Mike Bowen 编写了第 11 章，并为在该章中使用的游戏“超级泡泡兄弟”编写了客户端部分的代码。



Matt Bolt

Matt Bolt 写了第 16 章。Matt 为“古老家园”的“化身”创建了精灵序列图，他还为“古老家园”中旅馆里的 NPC 创建了精灵序列图。



Annika Hamann

Annika 为全书制作了 60 多幅示意图。



Robert Firebaugh

Robert 为第 11 章中的多人合作游戏“超级泡泡兄弟”创建了艺术形象，他还为坦克游戏制作了大桥。



Cyril Guichard

Cyril 创建了“古老家园”中的所有艺术形象以及绝大多数用户界面。Cyril 还为第 9 章中的坦克游戏创建了菜单屏幕。



Scott Smith

Scott 为第 11 章中的“超级泡泡兄弟”游戏编写了全部的服务器端代码。



Bruce Branscom

Bruce 为“古老家园”编写了地图编辑器的 AIR 程序，第 14 章将讨论它。

Tom Mcavoy

Tom 为在“古老家园”中购买家具物件编写了卖方用户界面，我们将在第 14 章讨论相关细节。他还编写了第 15 章中的好友列表用户界面的代码。

Mike Parks

Mike 撰写了第 4 章中有关 ElectroServer 管理的部分内容。另外，他还重新编排了精灵序列图的行序。本书中的一些屏幕截图也是由 Mike 制作的。

Jonathan Wagner

Jonathan 撰写了第 5 章中的聊天信息过滤部分，他还审阅了第 3 章。

Pat Makar

Pat 为“古老家园”配上了背景音乐。

Shannon Kozlowicz

Shannon 为第 13 章中的“化身”准备了分层动画文件。



Peter Royal

Peter 通过电子邮件给出了建议，并审阅了第 3 章。

Karl Prewo

Karl 创建了第 9 章坦克游戏中绝大部分的艺术形象。

Renee Sherbo

Rene 创建了第 5 章聊天范例中的用户界面。



Kelly Goodnow

Kelly 为“古老家园”地图编辑器创建了图标。

致　　谢

编写本书虽然耗费了大量的时间和精力，但仍不失为一大乐事。与我参与编写过的其他任何一本书相比，本书更算是合作的结晶。能有这么多才华横溢的人来帮助我创作这本数年来我一直想写的书，我真是太幸运了！

Mike Grundvig、Mike Bowen 和叛逆者 Matt Bolt 慷慨地承担起相应章节的撰稿工作。他们的劳动极大地提升了本书的含金量。特别要感谢 Grundvig，他为 Old World（古老家园）范例编写了完整的服务器端代码。

本书囊括了大量的精妙范例文件。感谢 Teresa Carrigan，她不仅为大多数范例开发了服务器端代码，还在书中各处用文字说明了如何使用这些代码。Scott Smith 极出色地为多人合作游戏《超级泡泡兄弟》（Super Blob Bros.）编写了服务器端代码，Robert Firebaugh 则为该游戏创造了极可爱的美术形象——谢谢 Robert! 当然同样要感谢 Cyril Guichard 为“古老家园”所做的精细的艺术设计。

Annika Hamann 将我画的那些学前班式的草图变成了清晰且引人注目的示意图，这真让人惊叹！

在少数几次我文不对题的时候（真的很少 ☺），“独行侠” Clint Little 与 Joel Stewart 的帮助使我能与他们具有洞察力的技术编审保持一致。谢谢他们！

谢谢 Wendy Katz 使我显得更聪明，还要感谢 New Riders 制作团队把一大堆文字和图片变成了极有条理（至少我自己这么认为）且外表美观的“阅读体验”。

我不得不感谢自然母亲，她带给南卡罗来纳州榆树城的绚丽春天使我得以在门廊上或者露台中撰写了本书的大部分内容。感谢我的妻子 Kelly，感谢她不得不数月以来忍受着在我的笔记本电脑屏幕的微光下入睡。说真的，如果我把屏幕从法式门那里转过去，你就看不到它了！

我无法忘记所有支持团队当中最大的那一群——动物。我们的猫“喵呜”的叫声总能让我觉得快活并使我保持清醒，我的毛毛狗 Free 拖着我去池塘散步让我能够时不时地活动一下腿脚，还有 25 万只蜜蜂——天啊！它们就是十足的魔鬼。我还不得不感谢那只孤独而叫声奇特的鸟，它在这段时间的出现总能引起我的兴趣。

最后要感谢我的母亲，是她赐予了我生命！

目　　录

第 1 章

网页游戏概述

简单地说，网页游戏就是托管在网站上且通过 Web 浏览器来玩的游戏。现在互联网上已有成千上万款这样的游戏，足见网页游戏的普及度之高。然而就在 10 年前，就算你天天都浏览网页，这样的游戏也是很少见的。现今有很多人已经迷上玩网页游戏了，有的人甚至一玩数小时而乐此不疲。

游戏玩起来很有趣，但更有趣的是编写游戏！

在本章中，我们将简要探究一下开发网页游戏的客户端技术与主要目标，以及这些要素和多人游戏之间的关系。

1.1 客户端技术

基于网页的游戏可以在数个不同的平台上实现（参见表 1-1）。这些平台提供编程语言并且还可以把代码编译成可发布到网上的游戏内容。要想通过网页和游戏实现交互，客户端的电脑上就必须安装名为“虚拟机”的程序，本地的 Web 浏览器通过它才能知道如何运行编译过的内容，Adobe 公司的 Flash 虚拟机也叫做 Flash Player。

表 1-1 开发网页游戏的最常见的平台

	开发者学习难度	运行时性能表现	虚拟机安装基本大小	开发者规模	开发难易度
Java	高	快	中	大	一般
Shockwave	中	快	中	中	一般
Unity	中	快	小	小	一般
Flash	低	一般	大	大	容易

尽管谈到功能和性能表现时，Java、Shockwave 以及 Unity 都要比 Flash 强大，但 Flash 却是创作网页游戏时最常用的平台。这是因为 Flash 更容易学习，它使艺术家和程序员之间的界限变得模糊，而且它的虚拟机在全球范围内有更广泛的安装基础。综上所述，Flash 是新网页游戏的首选平台。

在本书中，我们假设游戏和虚拟世界是通过 Web 浏览器和用户进行互动的。但也不一定，你可以从网页上下载 Flash 内容然后从硬盘上运行它们，这就不需要 Web 浏览器。Flash 内容可以被编译成 3 种不同的格式。

- SWF——SWF 文件可以被下载到用户电脑里而从硬盘上播放，但这并不理想，一般用户没办法完整地运行 SWF 文件，因为默认的网络安全设置会屏蔽掉那些用户感兴趣的交互行为，比如说和远程服务器对话。
- EXE——（只适用于 Windows 操作系统）我们通过调用 Flash Projector 可以把 SWF 文件编译成 EXE 文件，它内含 Flash Player，因此用户不需安装任何插件即可播放 Flash 内容。
- AIR——Adobe AIR 是创建可运行在用户本机上的 Flash 内容的最佳方案，用户必须安装 AIR 的运行时库（可以从 Adobe 网站上轻松地下载到它）。通过 AIR，Flash 可以完成一些通常情况下没办法完成的操作，比如往硬盘上写文件。

1.2　多人游戏适合的领域

尽管随着互联网的普及网页游戏也开始流行起来，但是在过去相当长一段时间内，多人网页游戏的发展，无论是从其数量上还是从精彩程度上看都比较缓慢。不是玩家们不喜欢玩多人网页游戏（他们乐于尝试任何新鲜事物），而是有许多因素制约着它的发展。作为一种极其普及的平台，Flash 尤其适合开发网页游戏。但是在过去，只有少数 Flash 开发者知道如何构建多人网页游戏。而这些开发者还可能会受限于没有适用的服务器端技术，并且没有足够的时间去开发游戏，结果当时几乎没有 Flash 多人网页游戏面世。

直到 1999 年 Flash Player 4 的发布，用 Flash 编写多人游戏才开始成为可能。我用轮询技术（详见第 2 章）写了我的第一个 Flash 多人游戏：国际象棋。接着，后续发布的 Flash Player 5 可以建立到远程服务器的 Socket 连接，这正解了我们的燃眉之急。于是大约从 2001 年起，包括我的公司在内的一些公司开始开发商业服务器来支持 Flash 多人游戏。

2006 年初，Flash 多人游戏开发终于进入快速发展时期。越来越多的用户需要它，越来越多的开发者也开始学习如何构建它。与此同时，一些基于 Flash 的虚拟世界也开始进行研发，出现了几个实验性项目。

2007 年和 2008 年，人们对 Flash 多人游戏和虚拟世界的需求激增，呈现爆发态势。

那么 Flash 多人游戏和虚拟世界目前在网页游戏领域中的发展态势如何呢？应该说如日中天！就我现今所见而言，几乎每个开发要求都会涉及多人互动组件。如今的态势是：提供多人互动内容的公司远远满足不了用户的需求。

1.2.1　典型目的

本节将把大多数的网页游戏归纳为几个大致的类别，以期向你阐述游戏开发的目的。这些

目的与本书其他内容没有必然联系，在此只作简要介绍。但是，你在设计一款游戏的时候应该牢记这些目的以便围绕它们进行设计。

在每一类别中，我们都探讨了多人互动内容对实现相应目标的作用。

1. 从广告条中获取收益

此项目的是通过广告展示获取收益。你需要尽可能多地吸引网民来访问你的网站，并且使他们留驻的时间更长，从而给你的站点带来更多的广告曝光次数。很多网站都采取游戏进行中在页面顶端加载滚动广告条的方式。而在最近几年里，越来越多游戏在你正式开始进入前的几秒钟里也开始加载页面广告了。

*多人互动内容如何促成此目标？*如果想让用户逗留更长时间，或许可以给你的网站添加聊天功能，从而让他们多停留一段时间。

2. 使站点更具吸引力

有时一些非游戏网站也会引入一些游戏以期让访客逗留更长时间。既然用户可以留在网站上玩这些游戏，那么当暂时不玩的时候，他们兴许会点击该站点中的其他链接。不过现在对这种做法的效果还存在着争议。

*多人互动内容如何促成此目标？*在这个特例中，我们的目的是使用户玩一会儿游戏然后浏览网站。既然目的不是长时间玩游戏，那么我们很难看得出多人互动内容能帮上什么忙。可能会有新途径让“培养”起来的用户和网站其余部分的游戏玩家互相沟通。

3. 市场营销

游戏经常被用来宣传推广电影、电视节目和体育赛事，还可以被用来推销日用消费品。我们公司开发的绝大多数游戏都属于这种类型。但因为游戏通常与其所推介之物的关系不是很大，所以有些时候这种方法还是要与使用广告条的方式相结合的。

*多人互动内容如何促成此目标？*对于市场营销来说，多人互动内容是大有可为的。有些站点只不过用了很简单的东西都能取得成功，比如 MTV 的 Back Channel（http://backchannel.mtv.com），它允许用户在观看电视节目时打字聊天。或者像 Mattel 的 Rebellion Race（叛逆赛车）游戏（http://www.hotwheels.com/games/rebellion/index.aspx），它通过实时的多人赛车游戏来推销公司的玩具车。

4. 推动下载

休闲游戏下载市场规模庞大并且非常成功，比如 Real Arcade（www.realarcade.com）和 Big Fish Games（www.bigfishgames.com）都通过免费的轻量级网页游戏来推销它们的可下载游戏产品。那些网页游戏是可下载游戏的简化版，如果你对网页游戏满意的话，你很可能会付费下载

那些正式版游戏。

多人互动内容如何促成此目标？这是目前唯一一个没有多人互动组件的领域，因为那些要用户付费下载的游戏多半都是简单的益智游戏，它可以让一个人独自玩上半天还不觉得烦。因此，我们很难料想多人互动内容在此领域内会有何建树。

5. 教育

如果你觉得游戏能令人愉快而教学则不是的话，那么这里的目标就是通过将教学和游戏结合起来使得枯燥的教学变得有趣。这可是个困难的任务，因为你可能会把游戏变得一点也不好玩——用户不会感兴趣（结果也学不到什么）。

多人动互内容如何促成此目标？多人互动内容的概念可以很大程度上运用于教育类游戏。有很多可能性，比如说通过竞赛调动求知欲，实时给予学习者帮助，或者提供一对一的个性化训练。

6. 提供订阅价值

通过支付包月费用，用户可以定期获得新特性与新内容。许多虚拟世界成功地运用这种方法，比如 Club Penguin（www.clubpenguin.com）和 Faraway Friends（www.farawayfriends.com）。

多人互动内容如何促成此目标？类似这种提供有偿订阅服务的网站大多是虚拟世界，而虚拟世界最吸引人之处则是社交。所以，在我们简单地定期为用户提供一些时鲜酷玩的同时，还可以为虚拟世界添加新的交互内容以使网友们交流起来更方便有趣。

7. 因为我能

人们开发游戏最常见的一个动机就在于此——开发者最初只不过想创造点东西而已。接着他们不断地修改与测试自己的作品，等到了一定程度后，他们可能会建立个人网站用以展示这些游戏或者把它们上传到共享游戏世界，比如 New Grounds（www.newgrounds.com）或 Kongregate（www.kongregate.com）。

多人互动内容如何促成此目标？如果目的是实验并创造出具有革新性的体验，然后为多人游戏开辟一条新路的话，开发者就需要尝试新潮的技术并要知道编程有什么乐趣和玩游戏有什么乐趣。

1.2.2　小结

Flash 多人游戏很好玩，市场需求量也很大，利用在本书中学到的内容并结合自身的创造力，你会取得惊人的成就！

第 2 章

连 接 用 户

想象一下你要开发的多人游戏吧！你希望它有多受欢迎呢？这款游戏可能会受到玩家的热烈追捧，你脑海中应该浮现出一幅熙来攘往、热闹非常的虚拟世界或游戏大厅的画面，不断有玩家加入或退出，大家互通有无，其乐融融。最关键的是，玩家对这款游戏的热情始终不减。

用户间是怎样获取对方信息的呢？他们是直接和对方联系，还是通过别的什么方式？这一章将为你解答这些问题并且告诉你对于交互我们所采用的一般处理方法。本章最后我们会选择一种将在全书范例中使用的技术。

2.1 连接技术

如果多用户间要发生任何交互的话，那么每个客户端都必须能收到其他客户端发来的信息，其发送的信息也必须能被其他客户端所接收。一般来说，处理客户端之间的交互有两种主要架构。

- **对等网络架构**（Peer-to-Peer，P2P）。信息只在客户端间传送，不需要服务器的介入（见图 2-1 左图）。

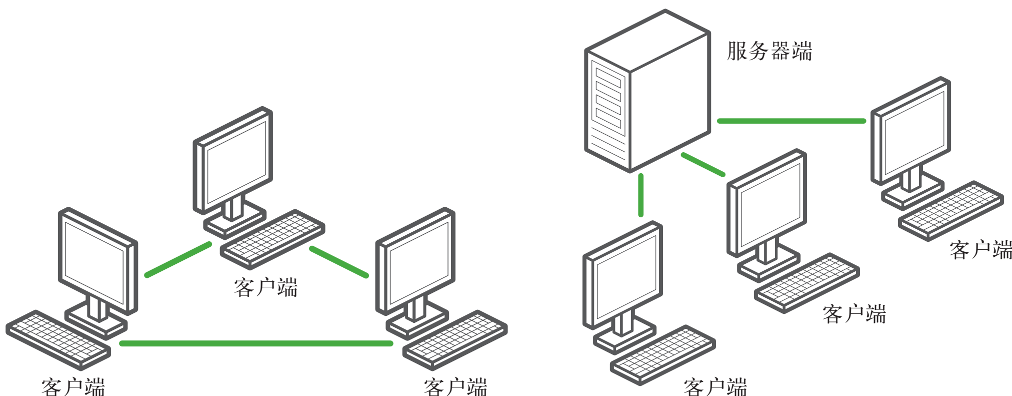


图 2-1 P2P 架构和 C/S 架构。在一个完整连接的 P2P 架构中，所有的客户端都彼此连接；在 C/S 架构中，客户端通过与服务器通信来交换彼此的数据

- **客户端–服务器端架构**（Client-Server，C/S）。客户端只把信息传送到服务器上，然后服务器再把信息传送给相应的客户端（见图 2-1 右图）。这种架构主要有两种实现方法：轮询（polling）和持续不断的 Socket 连接。我们一般采用第二种方法，即通过 Socket 服务器建立起持续不断的 Socket 连接。稍后再说明为什么要这样选择。

我们将在后面几节中介绍这些技术。

2.1.1　P2P架构

P2P 架构是两个或多个客户端不经过服务器而直接通信的架构。可能首先会使用服务器以让客户端间能查找到对方，但在此之后就不再需要服务器了。具体也分两种不同的形式：一种是完整连接拓扑架构，指的是每个客户端与其他每个客户端之间都必须有连接，信息可以直接在用户间交换；另一种是环状拓扑架构，指的是信息只有流经一个或多个客户端后才能传递过来的架构。本章中当谈到 P2P 架构时，指的是完整连接拓扑架构。



注意　在进一步讨论之前，必须首先说明一点：Flash Player 9 及更早的版本不支持 P2P 架构，从 Flash Player 10 起才开始有一些特性能支持它（稍后详述）。

一般来说，P2P 架构在游戏方面用得不多。很多情况下游戏似乎是在使用 P2P 架构，但实际上其中的一个玩家已被设定为主机来充当服务器的角色（在 2.1.1 节第 2 小节中详述）。然而，P2P 架构非常适用于搭建文件共享网络。通过它，网络游戏经常能够高效地为玩家们发放游戏补丁（比如《魔兽世界》），这不仅减轻了 Web 服务器的负担，而且也加速了玩家下载补丁的速度。

与 C/S 架构相比，P2P 架构技术有几个明显的优势，当然也存在几个缺点。先讲讲它的优点。

1. P2P 架构的优点

延时较小。延时就是指信息在从发出到接收这个过程中所用的传输时间。在 C/S 模式中，信息是先从一个客户端传到服务器端，而后再从服务器端传到另一个客户端。但 P2P 架构则是让信息直接在两个客户端间传递，这样就比 C/S 模式减少了一半的传递时间（见图 2-2）。



注意　延时越小，游戏的实时性越好。

不需要服务器端。既然 P2P 架构全是由客户端构建成的，因此也就没必要使用服务器端。这对于无论是游戏开发者还是发布者来说都是一件好事，因为他们不用再为维持运行游戏中央服务器而支付主机托管和管理费了。

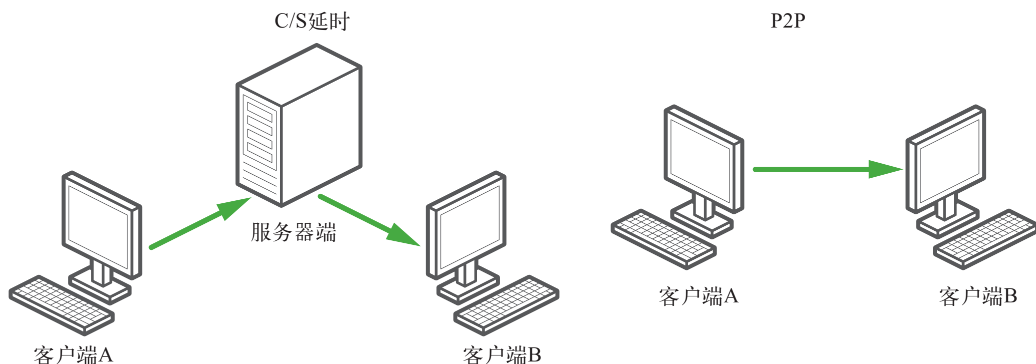


图 2-2 P2P 架构的响应时间要比 C/S 架构的响应时间约少一半

上面我们谈了对等网络架构的两个优点，下面我们来看看其不足之处。

2. P2P 架构的缺点

游戏的可扩展性差。当游戏的客户端相当少的时候，那就用 P2P 架构吧，因为它产生的延时很小且不需要使用服务器端。但是当游戏有很多客户端时，比如说成百或上千个客户端，那么 P2P 架构就不适用了。因为它需要每个客户端都和其他客户端保持一个独立的连接。那么如果有 1 000 个客户端的话，完整拓扑架构就要求每一个客户端都要建立并维持 1 000 个开放的连接，其中每一个客户端都要接受来自其他客户端发出的所有信息。而在 C/S 架构中，一个客户端只从服务器端接受经其智能筛选和归集后的信息。

另外，还要考虑一下此种网络的应用环境。假设你有 1 000 个客户端，但是它们全都在学校的子网络中，1 000 个用户彼此相连导致产生了 1 000 000 个连接，这将会使网络因不堪重负而瘫痪。但是如果同样数目的连接放到全球互联网上，则根本不会出任何问题。

争议解决机制不健全。假如有个双人 P2P 游戏，每一方都能操纵鼠标去吃奶酪。玩家 A 接近第一块奶酪，然后确定离它足够近时就可以吃了它，于是玩家 A 因为吃了奶酪而获得了一定的积分，然后他传递信息给玩家 B，告诉 B：这块奶酪已经被我吃了。玩家 B 接收并处理了这条信息后从自己的屏幕上删除了那块奶酪，并且更新了玩家 A 的得分。谁都没有异议，两个客户端对当前游戏的状态达成了一致。

但比如说现在玩家 A 和玩家 B 都在对下一块奶酪跃跃欲试。不巧的是这一次他们相互之间逻辑决策的时间就差了几毫秒，双方都各自认定是自己吃了这块奶酪，然后都给自己加上了分，然后互发信息给对方证明是自己把奶酪吃了。结果两个玩家就会互不相让。

所以，P2P 架构的一个缺点就是：没有中心逻辑决策者。还有一些范例更能说明问题。

- 两个战斗到最后的玩家间互发了一个必杀技，谁先倒下？

- 在赛车游戏中，谁第一个冲过终点？
- 玩家争着抢地上的钥匙，哪一位先抢到呢？

处理此类问题最好的方法就是采用 C/S 架构，让服务器来维护游戏运行中的状态，充当逻辑决策争议的逻辑决策者。

P2P 架构可通过选派一个客户端充当主机（但其本身还是 P2P 架构中的一个客户端）来解决逻辑争端，而这样做会带来下面两个新麻烦。

(1) 如果所有重要的逻辑冲突都要通过充当主机的客户端来解决的话，那就丧失了对等网络架构的延时较低这个优点，因为信息要从客户端传到主机由其处理完后再传回给客户端。

(2) P2P 架构游戏本身存在安全隐患，因为所有的重要逻辑行为都运行在客户端而没有在一个中央节点得到验证。那么如果你把所有重要的逻辑决策权都交给主机的话，无疑会使这个特殊的客户端在理论上具有了某种控制能力上的优势（作弊的机会大大增加）。

3. Flash Player 10 对 P2P 架构的支持

Flash Player 10 引入了一个新的通信协议：RTMFP（Real-time Media Flow Protocol）。此协议主要用于实现 P2P 架构的信息传输。这意味着开发者可以直接把一个 Flash 客户端和其他客户端连接起来。Flash 客户端必须首先连接到支持 RTMFP 协议的服务器上，比如 Adobe Stratus 服务器（Adobe 云服务器）；当然，在编写这本书时该服务器还处于测试状态。这个 Stratus 服务器的目的是帮助 Flash Player 的终端（Flash 客户端）相互连接。终端必须与服务器连接才能与服务器上的其他终端连接。

使用 Stratus 服务器，除了可以开发游戏外，Flash 开发者还能创建延时更低的语音或视频聊天程序。

2.1.2　轮询

轮询是一种没有实用价值的技术。我们在这里讨论它的目的是保持论述完整性以及指出它的不足之处。数年来我曾多次在多个留言板上见过开发者们尝试使用这种技术。的确，它在用户数目不大的情况下取得了一些成功——但只要负载过多就很容易崩溃。

轮询是指客户端定时给服务器端发送请求以检查信息更新的过程。拿聊天室举例来说吧，其中的客户端可能被设为大约每秒 2 次查询服务器以检查更新。客户端查询服务器端并且会得到响应（见图 2-3）。响应或者会以某种格式来指出不存在更新，或者会包含一条表明存在新信息的更新。新信息可以是用户已加入或离开聊天室，或者是已经添加了新的聊天消息。

乍看上去，轮询似乎并不坏。但是当我们开始进一步审视它并想象一下它的可扩展性时，问题就暴露出来了。首先，假如你有一个非常流行的万人同时在线的聊天室、游戏或虚

拟世界，假定每个客户端每秒对服务器查询两次，好吧，让我们计算下服务器要做多少工作：一秒两次……那就等于在 1 秒钟内会冒出来 20 000 次对服务器的查询以及服务器所做出的 20 000 次响应！这可是一笔庞大的带宽与服务器资源的支出，而且它是持续不断的，即使没有更新时也会如此。

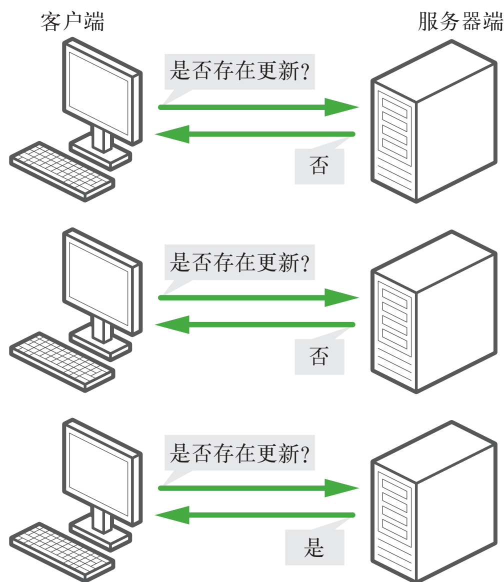


图 2-3　客户端轮询服务器以寻求更新，而一般是没有更新的

而实际上当请求在每秒数百次到 1 000 次时，系统性能就可能会出现问题。

在我们继续介绍之前还是先来谈谈轮询这种方法的两个主要不足之处吧。

首先，在事件发生和随后该事件被客户端接收到的这个过程中存在着延时。假如客户端每 500 ms 查询一次服务器，则当一事件发生后，用户端接收到此事件所用的时间为：500 ms 加上一般情况下的互联网延时。假设互联网合理延时为 100 ms，那么根据事件发生于查询周期中何种阶段而定，客户端将于事件发生后 100 ms ～ 600 ms 内接收到该事件。对于聊天程序或回合制游戏来说，这种延迟变化量是可以接受的，根本注意不到延迟。但对于其他类型的游戏来说很可能会出现问题。比如，在实时游戏中，如果对于每次客户端的请求，服务器的延时都在 100 ms ～ 600 ms 间变动的话，就会很难再玩下去。

另一个重大问题是客户端不在线情况下的处理。在轮询架构下，当客户端不再连接到服务器上时，服务器如何知道客户端消失了呢？唯一的方法就是让服务器去主动“注意到”客户端已不再发出任何查询请求。但是当服务器注意到某客户端消失之前，这种主动探查的方法就已经造成了高达几秒的延迟。

2.1.3　Socket服务器

Socket 服务器是经努力逐渐发展起来的一种技术。它是一种可于某处运行（一般是在远程物理服务器上）并监听连接尝试的程序。它接受并管理来自客户端的连接，智能规划客户端间的信息传送路线。Socket 服务器依托一个 IP 地址或主机名称，在至少一个端口上监听连接尝试（例如，主机名称为 electrotank.com，端口为 9899）。

客户端一旦与 Socket 服务器建立起连接，也就和服务器建立起了持续不断的 Socket 连接。通过这个开放的 Socket 接口，客户端能够将信息传递给服务器；反之亦然，而且根本不用请求！根本不用采用轮询或者其他类似的方式。因为这种 Socket 是开放并且随时可以使用的，所以信息相互传递就会非常快捷。

假如 Socket 服务器足够智能，系统中数据传输量就能最小化。有些服务器还可以集成客户端绑定的消息（关于其优点将在后续章节讨论）。集成消息是把多个消息打包后作为一个整体传输的一种消息。严格来说，P2P 架构中也能使用消息集成功能。有些商业 Socket 服务器产品（比如后面要谈到的 ElectroServer）已经具有了消息集成的功能，所以开发者们根本不用再编写相应的功能了。

为了对 Socket 服务器应用的事件驱动机制有个直观感受，我们来看看聊天程序的运行。如果在 5 分钟之内没有事件发生的话，客户端和服务器端之间就不会产生任何信息的传送，因此也就不会占用带宽。（对比一下轮询架构处理相同问题的方式吧：每一个与服务器相连接的客户端都经常要求服务器作出响应，不管有没有事件发生。）但当在聊天中发生了某事件后，比如说有用户进入或离开聊天室，或者聊天消息被发送，该信息就会被传送到所有连接到服务器需要接受信息更新的客户端上。

更新是基于事件驱动的而且是实时的。但在有些游戏中，互联网延时依然是一个待解决的问题。

2.2　可供选择的 Socket 服务器

在这里我们列出 3 种选择方案供你在考虑项目所需的 Socket 服务器时参考。我们会使用其中一种作为本书后续章节的主要方案使用。本书后续章节所展示的多人游戏概念并不是只适用于特定的服务器或执行方案，只不过当我们具体向你展示范例时，必须要有一个特定的工作平台。概念都是相同的，但实现的细节则根据所选的服务器而有所不同。

开发一个多功能、扩展性强并保持活跃的 Socket 服务器是需要很多开发者付出经年努力的。因此很多公司都宁愿选择已经开发好的服务器而不是自己研发。Socket 服务器的选择余地不是很大，绝大多数是商业性的。选择下列 3 个服务器之一，你就可以获得绝大多数你想要的性能，尽管在易用性、可扩展性和流媒体传输方面它们会有一些不同。

2.2.1　Adobe Flash Media Interactive Server

Adobe 提供的一些媒体服务器在功用上各不相同。其中一款服务器（www.adobe.com/products/flashmediainteractive）专注于提升连接客户端间的交互能力，而其他服务器则更注重于流媒体的传输。

我们将要关注的这款服务器可以让开发者自定义服务器端脚本以扩展服务器功能。它同样也支持传输高品质视频流和语音流，可以使用远端共享组件使连接的用户端上的数据更新同步。



注意　远端共享组件是一种数据容器，它可包含 Object、arrays、numbers、string、Booleans 以及其他一些数据类型的对象。客户端可修改它的属性，然后同步给所有能够接触到此组件的已连接客户端。

Adobe Flash Media Interactive Server 最多支持 5 个免费客户端连接，如果需要超过此数的客户端连接，则必须认证。关于如何认证，你可以去 Adobe 网站查询。

2.2.2　Red5

Red5（http://osflash.org/red5）是开源的服务器产品。它正努力朝着取代 Adobe 媒体服务器的目标而努力。Red5 支持几乎所有 Adobe 服务器所能提供的功能：流媒体传输、远端共享组件……而且它是免费的，你可以随意地修补添加它的功能，或者等着开发者去修补完善它。

2.2.3　ElectroServer 4

ElectroServer 4（http://www.electro-server.com）是 Electrotank 公司的产品，最多可以被 25 个同时在线的客户端免费使用，超过此数则需要认证。



注意　不要忘了，我是 Electrotank 公司的创办人之一，所以并不算一个公正的旁观者。

ElectroServer 被用于 Flash 游戏开发的历史要比其他服务器更久些，甚至超过了 Adobe 服务器产品。有相当多的虚拟世界和 Flash 多人游戏都是基于它构建的，这主要得益于其中那些经多年改进而成的便于游戏开发的种种特性与功能。它也支持语音和视频流，但却比不上 Adobe 服务器。

使用 ElectroServer 的主要优点之一就是它具有极大的可扩展性——它可以扩展到数十万个连接。

以上这些服务器都可以用来架构 Flash 多人游戏和虚拟世界，如果你打算使项目得以扩展，那就考虑一下服务器的可扩展性以及你的项目预算开支。

对于本书而言，毫无疑问，我们打算使用 ElectroServer 4，因为它的概念容易理解并且其 API 也较简单。

第 3 章

安全：你要面对所有人

其实，创建多人游戏或虚拟世界最终意味着要创建一种潜在的社会化游戏——你与全世界玩家相交互的游戏。这种社会化游戏唯一的规则就是：你永远不可能在对手尝试做出任何事情之前知道他们会怎么做。你所拥有的唯一防御手段就是在开发游戏过程中运用知识和发挥主观能动性来保护自己。这并不意味着你一定会输，而只是意味着你需要保持警惕并且要灵活应对将面临的安全挑战。换句话说，在多人游戏领域中“安全”这个字眼确实有众多语境和含义。

或许你认为你开发的虚拟世界不会受到黑客的攻击，因此决定跳过本章，但是，这样做你就大错特错了。你所创造的虚拟世界越是热门，就越是容易受到攻击。对许多人而言，乐趣并不在于游戏本身，而在于毁坏虚拟世界（其他玩家的乐园）这种行为所带给他们的挑战，破坏游戏规则就是他们的兴趣之源。

本章将就多种安全主题展开讨论，内容涉及从保护程序安全到保护儿童安全等诸多方面。

通过应用本章中所提供的知识，你将能显著地增强程序（虚拟世界、游戏或者其他程序）的防御能力以保护它不受恶意玩家的侵害。当程序功能失常时你将能够辨识出来并且知道如何对付大多数常见的攻击行为。有备无患，毕竟你不想输掉这场仗。

3.1 逻辑安全性

当程序员被问及安全性时，通常他们理解的是代码的安全性。而当家长被问及安全性时，你会得到完全不同的答案。本章中大部分的论述将是程序的物理安全性，但首先我们从另外一面入手，谈谈逻辑安全性（更确切地说，是为参与到虚拟世界中的玩家打造一个安全的环境）。

从这个角度来看，安全性和隐私是密切相连的。保持安全性的最好方法就是让参与到虚拟世界中的玩家使用匿名。然而此举通常会背离项目开发目标，因为它将极大地限制虚拟世界作为营销平台的用途。

然而，这个话题会牵涉到很多潜在的法律关系和议题，因此我们倒不如放弃对这些细节的关注。本节将会列出你需要了解的各种概念和事项，但不会为你提供具体建议。你应该去咨询

关于在线隐私问题方面的专家，他们会针对你的具体项目提供法律指导。

COPPA和数据收集

在美国，任何针对儿童在线行为安全问题的讨论都会涉及COPPA（《儿童在线隐私保护法》）这部法律。它是为了让孩子们安全上网而创立的。这部法律中绝大部分内容都直接针对（尽管并不只是针对）那些为孩子们提供在线行为（聊天、游戏及文章）服务的公司。

该法律主要想达成的目标围绕着数据采集与家长许可这两个方面而展开。一般而言最好避免采集可确认用户身份的信息。简单来说，即不要在账户的注册过程中询问用户的全名、住址和电话号码。虽然这只是一个范例，但通常你采集的信息越少也就越安全，而寻求法律咨询也很重要，那将确保你不会触犯法律。

聊天和交流

社交是虚拟世界中一个极重要的组成部分，但也被认为是一个巨大的危险因素。玩家间可以直接交谈的功能使虚拟世界更容易让人沉浸其中尽享其乐，但同时对此公开功能潜在的滥用也使得暴露在虚拟世界内的玩家容易受到侵害。家长们总是会担心自己的孩子在网络中结识坏家伙。

消除风险所最常用也最简单的方法就是取消公开式聊天功能，转而只允许“基于列表”的聊天功能。公开式聊天（open chat）指的是直接输入消息然后将它发送出去。基于列表聊天（list-based chat）则只允许玩家从一个包含可能会用到的消息的列表中选择并发送一条预制消息，因此也就保证了玩家不能收发某些“不合适的”信息。



注意　我们将会在第5章深入讨论这两种聊天类型。

而问题的症结在于：如何防止那些居心叵测之徒从其他玩家那里获取他们所不该知道的信息呢（包括真实姓名 、住址、电话号码在内的任何可确认用户身份的私密信息）？

如果虚拟世界允许公开式聊天的话，那么保护措施可归结为：聊天消息内容过滤、自我控制以及家长的许可。在一些虚拟世界中，家长通过管理面板来许可孩子们与某些人交谈。而在另外一些虚拟世界中，家长则根本不允许孩子们使用公开式聊天。在几乎所有虚拟世界中，信息过滤都是重中之重。

3.2　物理安全性

物理性安全指的是在实践中防止黑客盗取数据、毁坏程序和扰乱他人的用户体验的过程。有很多层级来保护游戏程序安全，不同层级会保护系统的不同部分。本节涉及了一般的安全性

规则与惯例，同时也探究了一些常见的对 Flash 程序进行的具体攻击方式。

3.2.1 问题与解决方案

我们力图解决的问题可以简单地概括成一句话：如何用 Flash 构建出一个能抵御网络攻击的虚拟世界？而其简单的答案让大多数人都不愿接受，那就是：简直不可能。虚拟世界只要在连接互联网的服务器上存在，面对网络攻击它就显得脆弱不堪。所以诀窍不在于构筑“马其诺防线”，而在于使得网络攻击无从下手。大多数网络攻击与利用都根源于一些简单的疏忽，如果你能改掉它们，就能立刻使虚拟世界变得更难被攻破。只需要一些预防措施和必要的技术，你就能确保虚拟世界相当地安全。

在本节的余下部分中，我们将探究构筑安全的虚拟世界和多人游戏所需遵从的基本原则和规定。你应该在创建新功能模块时通篇考虑这些原则并且要竭尽所能地实施它们。这样做就会显著地增强项目的防御力。

1. 最小化接触面

努力使得黑客所能接触到的系统区域最小化。这其中的道理非常简单：你要减少黑客能接触到的区域以防止他们利用漏洞来进行攻击。这个概念是全局性的，它适用于虚拟世界的所有相关事物（服务器、代码和数据）。

比方说你从服务器端向客户端传送用户数据，这些数据中有些是必需的，而有些则或许不是。但对于开发者们来说，把所有的数据都传送给客户端会是最轻松的做法。可是这样做太糟了——谁能保证你发送回来的数据中没有提供黑客攻击系统所需的最终关键数据呢？这样做不仅浪费带宽，而且还提供了可能会被用来对付你的数据，这完全是不必要的。

需要更具体的案例？那就瞧瞧游戏《亚瑟王的召唤》（Asheron’s Call）早期的做法吧。它总是把玩家周围大片地图区域内生成的可采集物品的全部信息都从服务器端传送给客户端，当玩家接近这些物品时游戏客户端程序就能正确地在场景中渲染出它们，但问题在于服务器端传送的信息所属地图区域过大，而不是只限于紧挨着玩家的一小块区域。这就存在一个漏洞——黑客们编制的一种工具使得所有生成的物品在小地图上的位置一目了然，于是它们就会很容易被寻获。这样一来黑客们就赢得了数都数不过来的物品，因为他们根本不用去寻找，物品只要一生成他们立刻就能知道其位置。

最小化接触面是很重要的，它能使得网络攻击的可能性微乎其微。所以请记住：如果不需要，就不要包含进去。

2. 晦涩不等于安全

这个常见的误解造成了太多的麻烦。仅仅使某些东西很复杂可并不一定能使它有多安全。

比如说加密得分时，你可不能认为只通过几次在得分上加乘一些数字以获得同样的结果就行了，这可不算是真正的加密。Flash 程序很容易被反编译。一旦有人反编译了程序，那么他利用你的算法进行逆向工程就能轻松地破解程序。应该保证即使黑客确切地知道了程序算法也不能破解，这样才真正安全。

3. 一概不信

如前所述，Flash 程序很容易被反编译或被逆向工程，所以在你编程时就应该假设黑客已洞悉 Flash 客户端的工作机制（比如怎样提交得分），因此你就必须保护客户端的安全。最好首先让服务器端帮助验证或处理数据。这方面的范例在本书中随处可见。

几年前由 Electrotank 公司开发的一个聊天客户端软件就是一个非常好的真实范例，它使用 HTML 来显示聊天文本，而且还会根据交谈对象以及聊天消息是否私密来使用不同的文本颜色。为了保证 HTML 格式良好，Flash 客户端在聊天消息被发送前对其进行了某种程度上的验证。

但这还不够！已经有黑客编制出了一种客户端，它能够在本地运行且去除了 HTML 验证检查。因为服务器端认为客户端已经做好了验证，所以能够让那些攻击文本顺利通过。接着黑客就在聊天室里制造了很多麻烦。他们把嵌入图片的 HTML 文件发送到文本字段中，或者发送能导引用户至不良网站的格式化超链接，或者把字号改得大得要命，要不就是把 HTML 格式变得很畸形以至于文本字段显示不正确。与其信任客户端的验证不如让服务器端对聊天消息进行验证，只有这样才能避免那些不良后果。

4. 验证一切

实际上，这和上一条原则是紧密相联的。不要信任客户端——你得验证一切由客户端发送给服务器端的东西，以确保它永不破坏规则造成麻烦。大体来说，验证是非常有效的，因为它还能协助你找出客户端 / 服务器端通信的错误。

关于验证的重要性，我们可以从一个战斗的范例中窥见一斑。假设在游戏中我的枪每秒射击一次。我每次射击都会传送一个消息给服务器端，然后服务器端会把这条消息广播给在我身边的所有人。当我每次扣动扳机时，客户端代码使用一个计时器来处理传送每一次射击消息。而黑客可以完全消除掉客户端每次射击后的延迟时间，这样他们就相当于有了一挺机关枪！解决这种问题的方法就是验证在一定时间内射击的次数，并且记录下在服务器端上发生的任何违例事件。

验证除了可以发生在服务器端以外还可以发生在客户端，认识到这点也很重要。在多人游戏中，客户端能侦测出从服务器端传来的错误消息并且能够将其忽略。稍后我们再介绍一个关于这方面的范例。

5. 诱捕一切

尽管简单的验证是朝着更安全的方向所迈出的一大步，但是你还应该编些防御代码。假设

客户端在错误的时间发送数据，或者发送的数据格式不正确。为此，你应该在服务器端诱捕这些错误并且尽可能详尽地记录下错误信息。这将成为你的“早期网络攻击预警系统”。记录中那些古怪的项目和错误将会有助于你根据需要修补或加固系统。

6. 非需则删

这条原则和上面所提到的最小化接触面原则是一脉相承的。默认情况下会有很多工具安装在物理服务器上。虽然这些工具提供给服务器管理员很多功能，但是对于管辖多人游戏或虚拟世界而言，大多数这些工具和它们所提供的功能都是可有可无的。如果你听任所有默认安装程序都留在服务器上的话，你可能就会把服务器暴露在攻击之下（即使不被攻击，这些程序也占用了不必要的空间）。黑客所需要的工具可能就在那儿——正因为你没有删除它。因此，如果你不需要它的话就将其删除。

这里要给你讲一个关于这方面的真实范例，它是我们几年前在使用 ColdFusion（一种服务器产品，原属于 Allaire，后来其东家变成了 Macromedia，现在则是 Adobe）过程中所经历的实事。ColdFusion 在安装时不仅包含有服务器而且还有一小组不错的范例文件。安装好 ColdFusion 后，这些范例文件就能被别人通过互联网自动获取。其中一个范例文件能够使那些经由浏览器且通过表单传递过来的恶意代码得以运行。精明的黑客不用费太大劲就能写出一些 ColdFusion 代码，这些代码能够下载并在服务器上执行一些能损害系统效能的程序。所以他们需要做的只是找到一个允许访问 ColdFusion 安装的范例文件的站点，给它们贴附上一段事先写好的脚本，然后眨眼之间，这个站点就被攻陷了！如果那些机器的管理员们事先干脆地删除掉成品服务器（production server）中的这些范例文件，那就不会发生这些问题了。所以假如你不需要这些产品的范例文件，那么就不要把它们留在产品中。

7. 不授人以柄

前文中我们已经强调了记录的重要性，现在让我们再回来讨论一下。记录对于你来说很关键，但它也会给黑客提供非常多的信息，所以无论如何都不能让黑客得到记录。具体实践中这将非常简单。

大多数高级 Web 开发语言特意在出现问题时提供详尽的错误信息，这些信息能使开发者轻松地找出毛病并予以修正。但是这也会透漏出太多的细节。你只需看一下默认的 ASP.NET 错误页面就会明白我的意思。所以，在产品中你应该一贯使用自定义的错误页面和错误信息。这既可以记录下产生的错误也使得终端用户无法了解原义。一石双鸟，你既能得到关键性信息又能保证其不被黑客所窥探窃取。

3.2.2　防火墙：有趣有利

此刻，你可能已经被上面的讨论搞得晕头转向，并想着到底应该从哪儿开始下手。幸而有

一种工具可以奇迹般地全面增强你的服务器安全性，那就是：防火墙。对于保证互联网应用程序的安全来说，正确配置的防火墙是很关键的。

尽管防火墙自身异常精巧复杂，但其使用原理却很简单。它的任务就是限制对其后方服务器的访问。防火墙为服务器提供的加固前端只包含几个特定的访问节点，从而限制了应用程序与外界的接触面。

实际上，虚拟世界防火墙的构造看起来有点像图 3-1 中这样。如你所见，防火墙阻断了所有针对服务器的外部访问或所有要在服务器（这里指的是数据库服务器）上运行的软件。因此这当然会更安全。

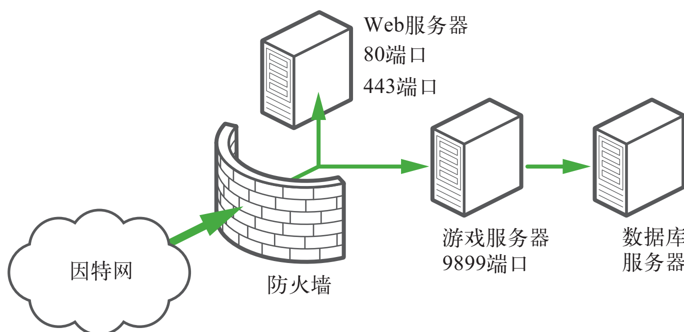


图 3-1

从图 3-1 中你还可以看出服务器只公开特定的入站端口。例如，Web 服务器只能在其 80（HTTP）端口和 443（HTTPS）端口被访问，而游戏服务器的访问则被限制在 9899 端口（默认的 ElectroServer 二进制端口）。

在现实当中，你可能会想要公开其他的一些端口，比如 SSH 入站端口和电子邮件出站端口，但是为简明起见，我没有在图 3-1 中标示出它们。即便如此，从理念上说还是要最小化进出环境的通道。这将极大地缩减留给黑客的可利用空间。

3.2.3 知己知彼，百战不殆

重要的是要知道大多数喜欢作弄你系统的人是为了要从中取乐。有时他们是那些想获取暴利的控驭型玩家，或者他们只是想做些没人能做的事。总的来说，他们感兴趣的是利用你的系统占点便宜，而不是一门心思地想使它瘫痪。

了解这些之后，你就知道他们会干哪种勾当及其所要使用的技术了。你要先把他们想得足智多谋。而你所拥有的优势之处在于，黑客们不得不从最终的游戏逆向工作并在摸索的过程中发现细节，而你仅需要预先做好安全计划即可。

在本节的余下部分中，我们来探讨下黑客针对应用程序最经常采用的攻击方式，以及他们特别针对 Flash 内容所采用的攻击方式。

1. 利用游戏行为

我们几乎可以确定，这是你首先能看到的针对你的虚拟世界的利用方式。设计中的缺陷或者未经深思熟虑就编写出来的功能都可能会被精明的玩家利用来获取一些在游戏中的收益。

关于这方面的一个范例是一个迷你型游戏。一种能减慢 CPU 速度的软件（类似 CPU Killer 那样的）可以将其变慢，这样就很容易取胜了。如果开发者没有限定（clamp）赢得游戏后所获得金钱的最大值，那么很有可能玩家通过一遍一遍地玩该游戏而获得大量金钱和很高分数——因为现在游戏变得很简单，很容易就赢了。



注意　“限定”是一个编程方面的术语，意为使数值保持在最大值与最小值之间。

一如既往，要阻止此类状况发生，我们首先要对参数值进行限定并且一开始就对输入进行验证。不过，还有一个特别的方法可以防止此类减缓速度的诡计得以实现，即使用基于时间的动画而不是基于帧的动画。如此可保证屏幕上的对象都能运动相同的距离，而与 CPU 的速度无关。

最后的修补措施就是测试游戏，以确保游戏不再含有容易被利用的漏洞。



注意　通过搜寻那些挣钱太多或者升级太快的玩家，你可以从中辨识出异常情况。观察他们的做法可以使你学会很多东西。

2. SQL 注入

从应用程序的角度来看，这无疑是所有可能的攻击方式中最恶毒的了。它也是最普遍的攻击方式之一。SQL 注入指的是黑客针对数据库注入并运行 SQL 语句的攻击方式。其原因多半是懒惰的开发者没有验证输入或绑定变量。

要想真正理解这一点，让我们来看一个具体范例。比如，在你的系统中有一个高分榜，选择一个指定用户的高分的代码是这样的：

```
select ScoreId, Score, Username, Date from Scores where
➔Username = '#name#'
```

假设 `#name#` 来源于未知的客户端。聪明的黑客就能简单地传递一条像这样的命令：

```
'; drop table Scores; --
```

因此最后实际运行的命令就变成了这样：

```
select ScoreId, Score, Username, Date from Scores where
➔Username = ''; drop table Scores; --
```

这样一个命令会把你整个的得分表格都删掉！这也意味着任何尝试访问此表的代码都将失效。这种类型的攻击既非常危险又很普遍，所以你必须时刻保护自己不受其害（不计任何代价）。

所幸实际上防止 SQL 注入式攻击并不很难。第一步是验证输入。这可是你无论如何都要经常做的事情，因此事实上它也不会给你带来额外的工作量。例如，如果想得到整型数值，你就要确保它不是字符串型。

另外你还要对所有输入数据都进行转义（URL 编码）。在此过程中你会发现很多命令都很有用，但最好的解决方法是使用*参数化查询*——在数据库系统被告知要查询的内容处对动态部分使用占位符，以此方式来进行查询。当你在运行时填充这些占位符时，字符的转义也就会被处理。使用预编译语句（prepared statement）不仅能为你带来更好的安全性，而且它们也能够提高性能，因为系统会将它们放于缓存中。

我们用来对付 SQL 注入式攻击的最后诀窍是限制它可能会造成的危害。在上例中，黑客能够删除掉数据库中的表格。这只会发生在一种情况下，即用来运行查询的数据库账户确实可以转储表格时。当然通常这主意可并不好——你得限制访问，只允许必需的最低级别的访问。或许你的虚拟世界里不需要配备那些在运行中能够添加并删除数据库对象（表格、索引或引用）的功能。假设是这种情况，你就应该给用户尽可能低的权限。在高度安全的系统中，不同账户用来访问系统中的不同部分，他们所能做的事也都有特定的 table- 和 command- 格式限制。

3. 跨站脚本执行

跨站脚本执行是针对 Web 应用程序的最常见的攻击之一。它的思路是让来自客户端的输入不经验证就显示出来。这将导致多种针对 Web 应用程序的肮脏伎俩得以实施。一个常见的跨站脚本执行的范例是这样的：一个论坛如果没有进行验证和清除特殊字符，那么包含在论坛回复贴中的一些 JavaScript 代码就可能将其劫持并把用户重定向到另一个网站。

这个问题在 Flash 程序中比较少见，因为 Flash 程序在多数情况下是不允许动态执行的。而值得关注的特殊之处在于那些用于字段格式化（field formatting）的 HTML 代码的使用情况。假如有一个聊天框，客户端发送了 HTML 格式的消息，那么很有可能因为在发送的消息中包含了无效的 HTML 代码从而使该消息在其他客户端上的显示发生错误。例如，客户端可能会把字体设得极小并且改变了字体颜色使其与聊天框的背景色一致，但却没有闭合字体标签（font tag）。这将导致随后所有聊天消息看起来都非常小而且颜色非常差劲。

要想解决这种状况只需简单地验证聊天消息并清除不需要的字符即可。如果清除了所有的 `<and>` 标记，那么 HTML 就不会被注入恶意代码了。

依据系统的设计，在接受方的客户端上也可以清除掉不需要的字符。如果文本传输没有加密，那么所有必要的 HTML 代码在接受时应该被添加，而接受到的那些显然无效的 HTML 则需

要被删除。对于让接受方客户端充当最后防线来说，这应该是一个不错的范例。

4. 数据包注入

简单地说，数据包注入指的是黑客往那些要传输到服务器端的数据流中注入含有他们想执行的命令的数据包的一种攻击方式。基本上这就可使他对他想要自动发生的行为进行编程。

让我们不妨假设你开发了一个实时射击类游戏，而且它也在玩家中变得流行起来。精明的黑客编写一个程序就能拦截所有在客户端与服务器端传递的消息，通过使用这样的方式他们得以建立一个游戏状态的内存中表现形式。在这种情况下，此程序的作用就相当于代理服务器：拦截所有东西并且将它们传递出去。不管任何时候只要有人在射程之内，此流氓程序就能自动地发送“射击”消息。听起来匪夷所思，但事实确实如此，过去有一些流行的第一人称射击游戏就被人采用数据包注入成功地攻击过。

有多种方法可以防止数据包注入攻击。首先，你得验证从客户端传过来的数据，保证它们是合法的——设法确保客户端不会太快地发送射击消息、穿墙而过，或者进行其他一些有违游戏规则（以及物理常识）的活动。永远不要在客户端上做出重要逻辑决策，它们应该总是在服务器端上进行。这样就能阻止精明的黑客发送那些拥有太多威力的信息——死亡消息、击中检测、游戏状态改变等。

确信只有有效数据在被传递后，你就需要开始强化协议本身了。一个行之有效的办法是当数据包在离开游戏与服务器端时自动对它们计数。这意味着如果有人试图注入新数据包，则会和服务器端所侦测的数据包数目相冲突，那么不管该包是否对游戏有用，系统都将处理掉它。在本书的范例中，这些工作都已经为我们做好了，因为 ElectroServer 会自动对进站与出站消息以及轨迹副本（tracks duplicate）进行计数。

安全性的另一个层面是控制消息的格式。如果黑客不能对消息进行逆向工程的话，他们也就不能复制这些消息。如前所述，晦涩并不等于安全，因此不要单指望一个复杂的协议可以帮你——你需要对协议进行加密。它不一定要多健壮，它只需很难被破解即可。TEA（微型加密算法）的多种版本或 XOR 都很有效，不过也存在其他许多可供选择的方案（ElectroServer 中内建的协议加密也能对此有所帮助）。

5. 修改消息

修改消息（与数据包注入方式源出同门）指的是黑客通过使用类似像 FireBug（一个 Firefox 浏览器插件）这样的工具或者一个“山寨”版的代理服务器（就像此前讨论过的那种）来修改从客户端传往服务器端的消息。黑客可能会修改一个游戏角色的运动消息来使其可以穿墙而过。他们也可能修改攻击消息以造成极高的伤害，或者校准目标使其精确度达到完美的程度。他们几乎无所不能。正因为这种攻击很容易就能实现，所以它也很普遍。

阻止修改消息的方法和阻止数据包注入的方法一样——对协议加密、在服务器端实施验证、逢重大逻辑决策时不要信任客户端等。

6. 内存变量攻击

编辑内存中的变量曾经是高级黑客的专利。但是随着新工具的涌现，这活儿变得容易了。许多文章都能教会任何一个人怎么去做到它，所以很遗憾，它也开始成为一种普遍的攻击方式。

大体的攻击过程是使用一个类似 Cheat Engine 那样的工具，然后告诉它要攻击的进程（这里指的是浏览器）。接着你搜寻一个已知的数值，比方说你在游戏中有 20 块金币，你就可以搜值为 20 的整数。这可能会返回数百个结果。然后你“锁定”这些结果，接下来在游戏中设法改变你所拥有的金币数量。比方说现在金币的数量降为 10，你就在原始搜寻结果中搜寻值为 10 的整数。到了这一步，你应该只搜得到几个结果了，这时你就应该知道改变哪个内存地址里的值能够影响游戏了。这种方法可以用来增加游戏得分、改变金钱数值等。如你所见，这将会使黑客在游戏中变得很强大。

幸而修补此类问题所采用的方法基本上和其他的一样：在多人游戏中，你只需要不信任客户端即可。验证客户端的输入，不要让客户端对逻辑做出重要决策。另一个能破解此方法的非常有效的措施是对内存中的变量进行加密。对重要变量进行加密后，黑客就无法搜索到它们，因为它们在内存中的存储状态不是“可寻”的。



注意　在 book_files/chapter3/MemoryCrypto.zip 中我们提供了一个能轻松实现该进程的 Action Script 3 程序。

3.2.4　关于安全性的最后说明

在本章中我们涉及了很多有助于保护虚拟世界安全的知识。多数都是基础性的，但良好的安全性正是由这些基础知识构筑起来的。你要保证你的虚拟世界的程序代码都经过仔细推敲并且拥有正确的验证方式（在处理客户端输入上你要稍微“偏执”些），这样你就能全副武装起来了。于恰当处实施加密再加上防火墙的保护，你就会拥有一个可认为是足够安全的能让玩家们乐享其中的虚拟世界。

第 4 章

介绍 ElectroServer

在第 2 章中，我们介绍了 Socket 服务器。它是一种通常在远程地点运行，并且通过互联网可访问到的软件，管理着数千个客户端程序（在这里我们指的是多人游戏和虚拟世界的客户端程序）间的相互通信。ElectroServer 是创建多人 Flash 交互内容时采用得最多的 Socket 服务器之一。

本章中，我们将介绍一些服务器的概念和特别针对 ElectroServer 的专用术语，同时也将教你如何安装 ElectroServer 以及如何编写简单的 hello world 程序。另外，我们还将考虑如何使用基于网页的管理系统来配置 ElectroServer。

最多可有 25 个连接用户（同时在线）免费且不受限制地使用 ElectroServer。你可在 http://www.electro-server.com/downloads.aspx 下载并安装它。

4.1 关于服务器的一些概念

本节中让我们来看看 ElectroServer 的一些概念和专用术语。其中多数都很流行且适用于其他服务器。通常在很多 Socket 服务器解决方案里都会出现这些概念，所以这里叙述的概念也有助于我们学习 ElectroServer 之外的其他服务器。

4.1.1 用户

用户指的是连接到（且登录进）服务器端的客户端。一个客户端可能会与服务器端建立不止一个连接，但其仍被认为是单个用户。因此我们要注意，尽管用户到服务器端经常只有一个连接，但他有可能会和服务器端建立起不止一个连接（见图 4-1）。例如，如果要从服务器端传递视频流给一个使用 ElectroServer 的客户端的话，用户就得另建一个用来处理语音 / 视频流的连接。



注意 “用户”这个术语贯穿于本书始末，但其含义稍显笼统。参照上下文语境，它可能是指连接到服务器端的客户端，此时它用来取代“客户端”这个术语；或者是指正在操纵客户端的人。

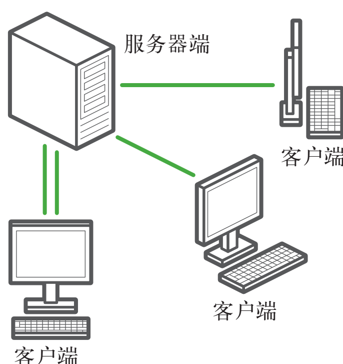


图 4-1　客户端连接到了服务器端，请注意，其中一个客户端和服务器端建立了不止一个连接

4.1.2　房间

房间是 Socket 服务器领域中的一个常见概念，它指的是用户的集合（如图 4-2 所示）。在 ElectroServer 中，借由房间这种载体，一个用户到多个用户间可以相互查看并进行互动。假如用户在房间里，他就可以给所有处于该房间内的用户发送聊天消息，然后该消息就会被广播给房间中的所有用户。这只是关于房间的一个简单用途。单个用户可以同时在不同的房间中。

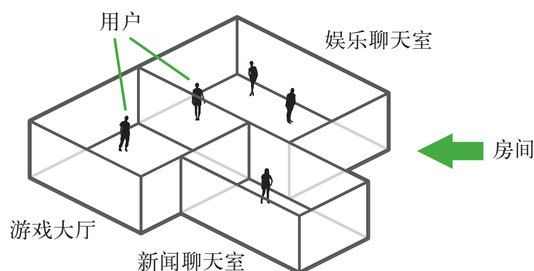


图 4-2　各种房间

在 ElectroServer 中有以下两种类型的房间。

- **固有型房间**。即使其内没有用户也将一直存在下去的房间。
- **动态型房间**。为单次使用所创建的房间。如果此类房间内的用户数目降为 0，则表明所有用户都已经离开该房间，那么该房间将被系统销毁。这也是最常见的房间类型。

房间有很多用途，最常见的两种就是促成聊天和聚众玩多人游戏。后续章节中我们再详细探究它的这些常见用途。

4.1.3　区

区指的是房间的集合。区这个概念非常有用，它主要被用来组织管理服务器上众多的房间。

区内的每个房间都必须有唯一的名称。区内房间中的每一个用户都能获得本区中的房间列表。每当房间列表变动时，用户都能自动订阅到更新后的房间列表。

如果房间列表很庞大或者非常活跃（服务器上不断地有很多房间被创建或被清除），那么已连接的用户每秒要接受很多次更新信息，以至于根本做不到与房间列表同步更新。使用更多的区有望解决这个问题（图 4-3）。



注意　不同的区里可以使用相同的房间名字。

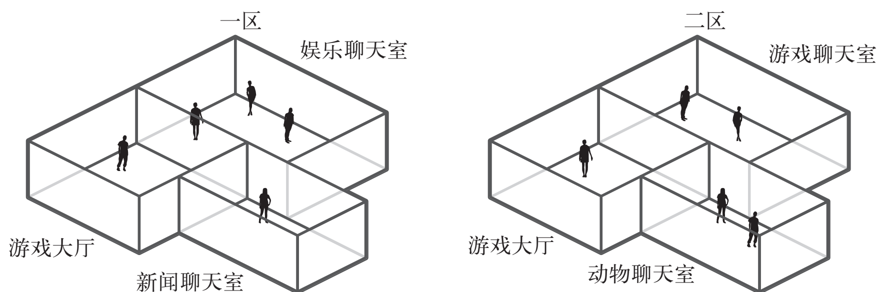


图　4-3

4.1.4　聊天

聊天是连接到 Socket 服务器上的用户们彼此间互动的主要方式。聊天信息是从一个用户发送给其他用户的文本。就像在其他 Socket 服务器解决方案中一样，在 ElectroServer 中也存在公开聊天消息和私密聊天消息。公开聊天消息指的是由用户发送给其所属房间内全体用户的聊天消息。通常公开消息最终会显示在客户端的文本字段中，如图 4-4 所示。



图　4-4



注意　关于“聊天”这个主题，我们将在第 5 章更详尽地论述。

私密聊天消息指的是由用户直接发往另一个或多个用户的消息（图 4-5）。与公开聊天消息不同的是，目标用户并不需要和发送方处于同一房间，就此而言，甚至他可以根本不在任何房间中。通常私密聊天消息是由单独一个用户发送给另一个用户的，比如说聊天室内用户间的私下交谈或者属于游戏中同一团队的玩家间的秘密交流。在多人游戏和虚拟世界的多数情况下，只有好友之间才会互相发送私密消息。

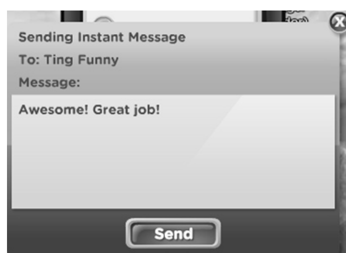


图　4-5

4.1.5　好友

社交网络是典型的多人体验中的一个庞大的组成部分。除了在一起玩游戏和竞赛之外，玩家间还喜欢建立起一种超出单纯娱乐时段存在的关系。他们彼此间会乐于把对方的网络身份标示为“好友”（buddy），并且将其添加到好友列表中。当用户以后重返多人互动程序时，她只需查看其好友列表就能知道哪些好友在线而哪些离线（见图 4-6）。用户就可以给当前在线的好友发送消息甚至邀请他们一起来玩游戏。

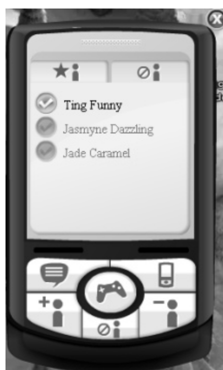


图　4-6

在有些虚拟世界或社交网络平台中，即使用户的一位好友当前处于离线状态，该用户也可以给其发送聊天消息。当这些离线用户重返该网络时，他就能接收到别人传给他的延迟了的消息。

4.1.6　EsObject

这是我们介绍的第一个真正的 ElectroServer 专有概念。乍看上去，EsObject 似乎有点怪异，但当你熟悉了之后，你就会觉得它非常有用。

EsObject 是一种以相同结构存在于服务器端和客户端的类。先创建该类的新实例，接着将数据贮存到该对象上。由于事务处理层（transaction layer）知道如何将对象序列化和反序列化，于是 EsObject 对象就可以轻松地在客户端和服务器端间进行交换。

序列化与反序列化

对象被序列化意味着其数据被表示成另一种结构形式以用于存储或传输。比如说下面这个对象：

```
var person:Object = new Object();
person.name = "Jobe";
person.age = 33;
```

此对象可以被表示成像下面这样的 XML 形式：

```
<person>
     <name>Jobe</name>
     <age>33</age>
</person>
```

这种将数据对象转换为一串 XML 表示方式的操作被称为“序列化”。数据对象还可能被转换为二进制或其他多种格式，序列化后的数据再被转换为原来的数据对象的过程被称为“反序列化”。



注意　事务处理层指的是解析并格式化在客户端与服务器端间相交换的数据的客户端与服务器端 API 部分。它会读取数据然后对其进行格式化，接着通过 Socket 服务器将格式化后的数据发送出去，而后它又通过 Socket 服务器接受数据，将其解析并装载进可用对象中，最后派发一个能够被特定应用程序代码捕捉到的事件。

EsObject 对象自始至终是通过 API 来被用于客户端与服务器端的通信的。下面就是一个用 ActionScript 创建 EsObject 对象的范例：

```
var esob:EsObject = new EsObject();
esob.setString("name", "Jobe");
esob.setInteger("age", 33);
esob.setStringArray("petNames", ["elfie", "bosley", "clyde"]);
```

上例中我们先创建了一个 EsObject 类的新实例，而后为它添加了数据。添加到 EsObject 对

象里的每一个属性值都对应着一种严格的数据类型。上例中已经涉及了 `String`、`Integer` 和 `String Array` 数据类型。

下面的列表完整地列出了为 EsObject 所支持的数据类型。

- String/String Array
- Integer/Integer Array
- Number/Number Array
- Boolean/Boolean Array
- Byte/Byte Array
- Character /Character Array
- Double/Double Array
- EsObject/EsObject Array
- Float/Float Array
- Long/Long Array
- Short/Short Array



注意 最常用到的数据类型是 String、Integer、Boolean、Byte 和 EsObject。

你开始时可能会感到疑惑：为什么要将所有的数据都放入一个 EsObject 对象中呢？这会是个好主意吗？看上去这似乎要写很多代码并且不得不确认那些似乎过分严格的数据类型。但是，它实际上会在以后为你节省时间并减少麻烦。通过使用 EsObject 和严格的数据类型，你可以减少代码中的歧义并且最终会使代码更易管理。

使用 EsObject 的另一个优点（明显但容易被忽略）是带宽。对 EsObject 对象的序列化进程（你看不到）所产生的数据包尺寸是极小的，因此也能保持最小的带宽。

4.1.7 扩展

除了提供具有高可扩展性的连接层之外，大多数实用的 Socket 服务器首先会提供基本级别的即开即用功能，如房间、聊天、好友以及其他一些功能。但如果多人游戏或虚拟世界所需的一些特殊功能或特性没有被包含在你现在所使用的服务器的核心功能集合中该怎么办？一个好的 Socket 服务器会提供给你方法让你自己添加这些特性。通过使用自定义编码来扩展服务器的功能，你就能实现你想做 的任何事。

ElectroServer 和其他许多服务器都支持一种叫做扩展（extension）的程序。扩展就是在服务器上运行的提供了服务器非内建功能及特性的自定义代码。

扩展可以包含一种或多种类型的对象用于扩展 ElectroServer 的某些功能。它支持 3 种用于

扩展并增强 ElectroServer 的对象：事件处理器（event handler）、插件（plugin）和托管对象工厂（managed object factory）。每种对象都可以包含变量用来定义在对象创建时应被赋予的值。

1. 事件处理器

事件处理器允许开发者使用一些自定义逻辑来作为事件的结果。尽管也能用 ActionScript 1，但通常它们是用 Java 来编写的。

ElectroServer 目前允许为以下事件类型创建事件处理器：

- Login——用户登录 ElectroServer；
- Logout——用户登出 ElectroServer；
- User Variable——用户已更新了用户对象附属的 EsObject 对象；
- Room Variable——用户已更新了房间对象附属的 EsObject 对象；
- Buddy List——用户已更新其好友列表（添加 / 编辑 / 删除）；
- Private/PublicMessaging——用户已给其他实体（房间 / 用户）发送了信息。

让我们拿 Login 事件处理器为例。客户端连接到 ElectroServer 并提供其登录凭证，其姓名和密码就会通过与数据库对照进行验证，然后事件处理器来决定是否接受或拒绝用户的连接。



注意 几乎所有使用 ElectroServer 的大型程序最终都会使用自定义编写的 Login 事件处理器通过比对数据库来验证用户凭证。

2. 插件

插件（通常用 Java 来编写）提供了那些核心特性所未提供的必需的可扩展功能。客户端可以和插件进行对话，插件也可以相互间进行对话。ElectroServer 提供两种类型的插件：房间级插件和服务器级插件。

房间级插件负责管理任何可能需要附属于某一特定房间的功能。它们往往被用于处理游戏逻辑和附加的房间功能。关于房间级插件的一个很好的范例应该是纸牌游戏：插件会处理所有发牌的规则、哪个玩家得到牌、计算得分以及决定谁是赢家。创建一个房间级插件且将其作用域限定在一个房间中，这样，它就成为了一个实例。客户端只可以和任何作用域为其当前所属房间的房间级插件对话。

服务器级插件扩展了 ElectroServer 的执行能力。这些扩展功能所涉及的方面从全局性地增强房间功能一直到为想要改变游戏某些方面的功能而公开一个外部接口。和被需要时才被创建的房间级插件所不同的是，服务器级插件只被创建一次并将持续存在。客户端能够和任何服务器级插件对话。

关于服务器级插件的一个很好的范例是访问远程 RSS 源。如果你的虚拟世界需要在其中的某些方面显示新闻条目，那么服务器就需要加载新闻。在这种情况下，明智的选择就是让服务器级插件加载并管理这些最新消息以便让任何其他插件在需要的时侯访问它。

3. 托管对象工厂

托管对象工厂允许创建那些需要长久保持活跃的对象的实例。我们使用此类工厂创建的一种常见对象即为数据库链接（database connection）。它允许定义其属性和数据库类型以连接到插件并从其中检索数据。

让我们来看看下面这个基于 MySQL 的托管对象的创建过程吧，它是为一个基于 Java 的插件而编写的。

```
EsObject esDB = new EsObject();
esDB.setString("poolname","mysqlpool");
Connection c = (Connection)getApi().acquireManagedObject
➥("ManagedDBConnection", esDB);
```

4.2 安装 ElectroServer

ElectroServer 最大的优势之一在于它支持多种平台。在本节中，我们将介绍它在几种支持平台下的安装方法。本章中所有的安装说明都是基于我们所使用的 ElectroServer 4.0.6。



提示 在 www.electro-server.com/ 中可获取到最新版的 ElectroServer。



提示 所有安装方式都要求你的 Java 虚拟机必须是最新版的（截止撰写本书之时它是第 6 版的第 13 次更新），到 http://java.com/en/download/ 可以下载最新版的 Java 虚拟机。

4.2.1 Windows系统下的安装

以下安装说明适用于 Windows XP 和 Windows Vista 操作系统。

(1) 从 ElectroServer 网站下载并运行 ElectroServer_4_0_6_Windows.exe 安装包。

(2) 依照提示操作直到出现 Select Components（选择组件）提示为止。勾选每一项，因为这些选项可以为你提供更多的本书范围之外的阅读资料以及范例文件。

(3) 继续依据提示进行安装，保留每一项的默认设置即可。当出现 Select Server Mode（服务器模式选择）选项（见图 4-7）时，选择 Professional mode（专业模式）。

(4) 在弹出 Web Server（Web 服务器）提示框（图 4-8）时，把上面的默认值记下以备将来参考，然后点击下一步。如果愿意的话，以后还可以再修改这些参数值。

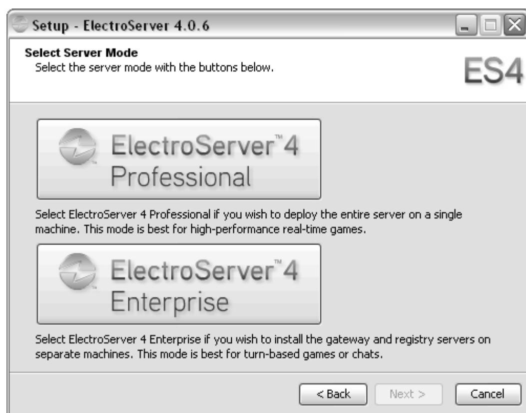


图 4-7

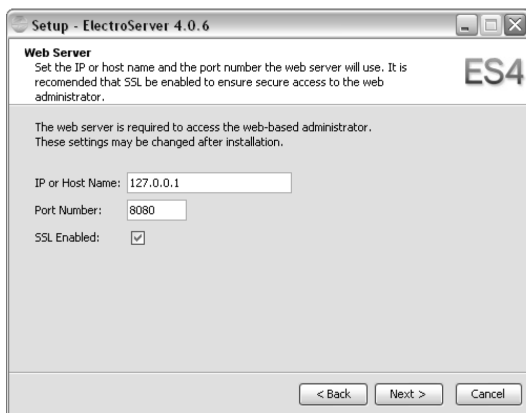


图 4-8

(5) Administrator User（管理员用户）提示处保留默认值即可。默认的管理员用户名为 administrator，密码为 password。如果想共享此服务器的话，你就得修改它们。你以后可以改变默认用户和添加额外用户。现在让我们点击 Next 按钮。

(6) 当被问及是否安装为一项 Windows 服务时，确认不勾选此项，点击 Next 按钮，然后在 ElectroServer 安装时你就可以喝咖啡了。

(7) 安装结束，点击 Finish 按钮后就可以使用新安装好的 ElectroServer 了。

(8) 进入 Windows 操作系统的 Start（开始）菜单，依次进入 Programs > ElectroServer 4.0.6。从这个菜单中，选择 Start ElectroServer。

ElectroServer 会以控制台窗口方式运行，它将输出所有 ElectroServer 需要显示给你的消息。

4.2.2 Linux/UNIX系统下的安装

以下安装说明适用于 CentOS、Fedora 以及其他支持 RPM 的 Linux 分发版。

(1) CentOS、Fedora 或者其他支持 RPM 的 Linux 分发版选择 Electro_ 4_0_6_linux.rpm 安装包。其他 Linux 分发版选择 Electro_4_0_6_unix.tar.gz tarball 安装包。

❒ 想要安装在 CentOS、Fedora 或者任何其他支持 RPM 的 Linux 分发版上的话，既可以使用系统中的安装包管理器，也可以按照如下的命令行来进行安装：

```
rpm -i ElectroServer_4_0_6_linux.rpm
```

❒ 如果想要安装在其他 Linux 分发版上，那么既可使用你所选的解包器将其提取出来，也可以使用如下命令行来进行提取：

```
tar -xvzf ElectroServer_4_0_6_unix.tar.gz
```

(2) 当你解开包后，ElectroServer 运行前的准备工作就完成了。所有在 Windows 系统下安装时需要保留的默认值也适用于此，因此要配置的选项其实很少。

(3) 找到你新安装的目录：

- 如果你用的是 tarball 安装包进行解包的话，那就进入 ElectroServer_4_0_6 目录；
- 如果你已经安装了一个支持 RPM 的 Linux 分发版，那就进入 /opt/ElectroServer_4_0_6 目录。

在正确的目录下，你会发现 3 个 Shell 脚本：ElectroServer、GatewayServer 和 RegistryServer。

(4) 使用下面的命令来运行 ElectroServer 脚本：

```
./ElectroServer
```

ElectroServer 现在应该从控制台窗口中启动运行了。控制台窗口会为你显示任何 ElectroServer 可能需要告诉你的消息。

4.2.3 Mac OS X系统下的安装

对于 Mac OS X 系统的安装，可以参照在不支持 RPM 的 Linux 系统下的安装说明，不过要注意，如果你的 MAC OS X 系统版本低于 10.5 的话，你就得对它进行升级以利用 Java 虚拟机第 6 版第 7 次更新或其更新版本。

4.3 编写 hello world 程序

还有什么程序会比 hello world 更适合作为你编写的第一个程序的呢？我们是该敲点儿代码了！

本节将为你介绍 ElectroServer API，并且使用它来连接到 ElectroServer，登录并发送一条私人聊天信息。

4.3.1 ElectroServer API

ElectroServer API 是一种 ActionScript 3 API，多人游戏程序利用它来连接并与 ElectroServer 通信。这种 API 以 SWC 文件的形式来提供，你能在本书中所有那些和 ElectroServer 通信的范例中 lib 目录下找到它。



注意 第 5 章将举例说明绝大多数的由这种 API 自动管理的数据。

SWC文件

SWC 文件是在开发时用到的预编译代码。它能包含代码或像用户界面元素那样的二进制资源。只要它被放置在程序的类路径中，SWC 文件就能一直被该程序使用。

ElectroServer API 允许客户端与服务器端建立连接并随后与其通信，它也能执行一定量的数据自动管理。比方说，API 会记录下当前你所在的房间以及在这其中每个房间内的用户列表。

主要存在 3 种可以在服务器端与客户端间循任一方向传递的消息。

- **请求**——一种由客户端创建然后被发送到服务器端的对象。比如，当你要登录时，要创建一个登录请求，然后将其发送给服务器端。不管出于什么意图和目的，所有由客户端发送给服务器端的对象都是请求。
- **响应**——由服务器端创建的作为请求结果发送给客户端的对象。比如，一旦服务器端接收到客户端的一个登录请求，它就会处理该请求然后生成登录响应。响应将包含诸如登录是否成功这样的信息。
- **事件**——一种不一定由请求触发的由服务器端发送给客户端的消息。比如，在聊天室内接收到一条聊天消息是一个事件，这种聊天消息可能是由同一个用户、其他用户甚至是由服务器端本身产生的。

你能用 API 所做的大多数事情都是由请求所驱动的。客户端创建了某种类型的请求对象，接着把信息填充给它，然后再把它传给服务器端。



注意　在本书中只要我们一遇到 API 调用就会予以讨论。你可以在 ElectroServer 网站 www.electro-server.com/documentation.aspx 中找到详尽的文档。

4.3.2　编写你的第一个聊天信息

在本节中我们来看看以下这个范例，它在本书范例源文件中的位置是 book_files/chapter4/hello_world。

打开项目文件，使用项目标签浏览并打开 Main.as 类文件。本项目只使用这一个类文件，而且编译此项目也不需要什么可视化资源，你唯一可以看到的东西就是由 ActionScript 创建的文本字段。

Flash Develop

跟所有本书中的范例一样，Hello World.as3proj 文件是一个 Flash Develop 的 ActionScript 3 项目文件。你可以在 www.flashdevelop.org 下载 Flash Develop（一个免费的 ActionScript 代码编辑器）。

让我们编译这个程序看看出现了什么。确认 ElectroServer 在运行。你看到的应该像图 4-9 这样。

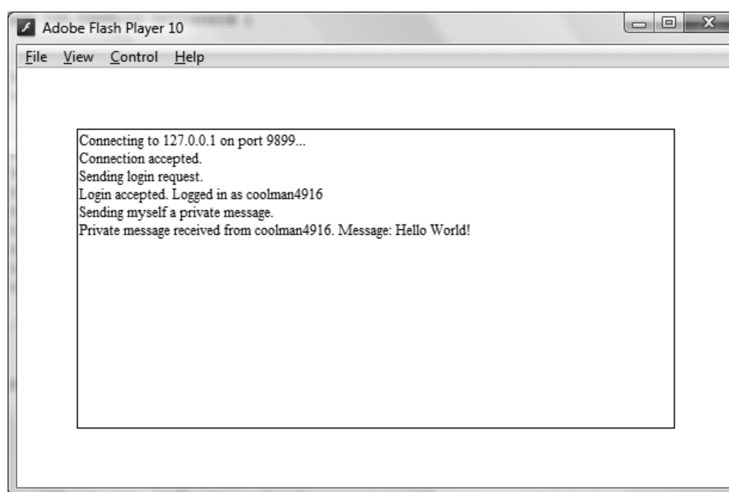


图　4-9

编译过的程序是按照下面的步骤来运行的：

- 准备一个要用到的 ElectroServer 实例；
- 在 IP 地址 127.0.0.1 和端口 9899 处连接 ElectroServer；
- 以随机的用户名登录；
- 给程序本身发送一条私人聊天消息并且捕捉这条消息。

程序会将每一步作业记录都输出到屏幕上的文本字段中。

1. 准备 ElectroServer 实例

在连接 ElectroServer 之前所做的第一件事就是要创建一个 `ElectroServer` 类的新实例。`ElectroServer` 类是通往整个 API 世界的大门。

在 `Main` 类的第 36 行标示出了如何创建 ElectroServer 的新实例。

```
_es = new ElectroServer();
```

因为我们以后会在其他程序中使用这个类实例，所以它是一个类属性。

现在，ElectroServer 的实例创建好了，接下来我们向它添加事件监听器以便捕获由服务器端所返回的响应及事件。

```
_es.addEventListener(MessageType.ConnectionEvent,
➔ "onConnectionEvent", this);
_es.addEventListener(MessageType.LoginResponse,
➔ "onLoginResponse", this);
_es.addEventListener(MessageType.PrivateMessageEvent,
➔ "onPrivateMessageEvent", this);
```

第一行代码为连接事件创建了事件监听器，当连接发生或超时时，就会执行 onConnectionEvent 函数。同样，接下来两行分别为登录与私人聊天消息创建了事件监听器，我们在后文中再讨论它们。

addEventListener 函数的每一个参数都值得讨论。首先是 MessageType 类。这个类用于存储可能和 API 有关的每一种请求、响应及事件类型的引用。当添加事件监听器时，你要使用 MessageType 类来指明你要捕获的响应或事件。

第 2 个参数是一个字符串，它包含当事件发生时你想要调用的函数的名称。这一项对于 ElectroServer API 来说并不理想；因为要支持以前的客户端，所以此 API 的代码基础是维持在 ActionScript 2 上的，而 ActionScript 3 向上类型转换的代码却习惯于生成此 API 的 ActionScript 3 版本。在处理过程中为添加事件监听器而不得不使用字符串式函数名称，这虽然是一个令人遗憾的副作用，但是它除了缺少编译时检查所用函数名称这一点外，并不会降低代码的执行速度，也不会出现任何其他问题。

第 3 个参数描述了函数存在的作用域。

2. 建立连接

既然有了 ElectroServer 类实例并且也为其添加了正确的事件监听器，那么我们就能实际地做点什么了——建立连接！ElectroServer 能支持数种不同的网络协议（文本、二进制、HTTP 和 RTMP）。本书通篇采用二进制协议。它既轻巧又迅速因此成了我们的不二之选。

首先，在代码第 44 行告诉 ElectroServer API 我们想使用的网络协议：

```
_es.setProtocol(Protocol.BINARY);
```

接下来，在代码第 47 行尝试连接 ElectroServer：

```
_es.createConnection("127.0.0.1", 9899);
```

假设目前 ElectroServer 正在运行，并且使用了所有的默认设置，那么 ElectroServer 就会监听在端口 9899 处的本地 IP 地址为 127.0.0.1 的 Socket 连接。

无论连接成功与否，onConnectionEvent 函数都将被调用：

```
public function onConnectionEvent(e:ConnectionEvent):void {
  if (e.getAccepted()) {
       log("Connection accepted.");

       //build the request
       var lr:LoginRequest = new LoginRequest();
       lr.setUserName("coolman" + Math.round(10000 *
       ➔Math.random()));
```

```
        //send it
        _es.send(lr);

        log("Sending login request.");
    } else {
        log("Connection failed. Reason: " + e.getEsError().
        ➔ getDescription());
    }
}
```

注意一下该函数所接受的 ConnectionEvent 参数。所有事件与响应都接受一定类型的对象作为参数，该参数包含了所出现的信息。

另外，从上面的 if 语句中可知这个事件对象包含一个确定连接成功与否的布尔值。如果连接没有成功，我们将把它记录下来，接着会记录下连接为何失败的错误描述。

如果连接成功，就将其记录为成功，然后尝试登录进服务器端。

3. 登录

默认情况下 ElectroServer 允许用户以任意名称登录，但此名称必须是唯一的（不存在重名）且通过了作为 ElectroServer 程序一部分的默认粗俗语汇测试。（如果以粗俗不堪的词汇作为用户名登录，系统会报错。）

要登录 ElectroServer，必须先创建一个 LoginRequest 对象并为其添加一个用户名（本例中使用随机名称），然后把此对象发送给服务器端。总而言之，所有的请求都是这样处理的：首先创建请求，然后添加进信息，最后被传送到服务器端。

接下来服务器端处理请求并返回一个登录响应，此响应在 onLoginResponse 函数中被捕获：

```
public function onLoginResponse(e:LoginResponse):void {
    if (e.getAccepted()) {
        log("Login accepted. Logged in as " + e.getUserName());

        //create the request
        var pmr:PrivateMessageRequest =
        ➔ new PrivateMessageRequest();
        pmr.setUserNames([e.getUserName()]);
        pmr.setMessage("Hello World!");

        //send it
        _es.send(pmr);

        log("Sending myself a private message.");
    } else {
        log("Login failed. Reason: " + e.getEsError().
```

```
        ➔ getDescription());
    }
  }
```

如果登录被许可，那么 LoginResponse 对象的 getAccepted() 方法返回值为 true。如果登录没有被许可，那么我们将记录下此信息，并且记录下有关为何登录失败的错误描述。

假如成功登录，就给自己发送一条私密消息。

4. 发送一条私密消息

要想用 ElectroServer 发送一条私密消息，首先你得创建一个 SendPrivateMessage-Request 对象 , 接着，诸如接受方用户名列表以及加载的消息之类的信息就被添加到该对象中。在此范例中，用户名列表是个只有一个名字（其实就是我们自己的）的数组。

服务器端处理请求并最终会给列表里的每一个用户发送私密消息事件。本例中的私密消息事件会被 onPrivateMessageEvent 函数所捕获：

```
public function onPrivateMessageEvent(e:PrivateMessageEvent):
➔ void {
  log("Private message received from " + e.getUserName() +
➔ ". Message: " + e.getMessage());
}
```

在这个函数中我们只是简单地记录下：收到消息，消息发送方及消息本身的内容。

4.4　管理面板

既然已经成功地创建了第一个客户端程序，你应该准备好将其与 ElectroServer 连接了。但是在此之前，你还必须确认 ElectroServer 的设置确实如你所需。在本章末尾，我们来介绍一下管理面板（administration panel），它允许配置 ElectroServer 的许多选项。

在管理面板中，你可以定义 ElectroServer 用以监听 Socket 连接的 IP 地址和端口，也可以管理扩展（extension），配置服务器端所允许的用户数量，甚至还可以创建语言过滤器来屏蔽掉聊天信息内的淫秽词句。



注意　*以防这一点并不明显，需要运行 ElectroServer 来使用管理面板。*

在运行 ElectroServer 后，打开浏览器，输入管理面板的默认地址 https://localhost:8080/admin/，你就会看到图 4-10 所示的屏幕显示。

输入在安装进程中你提供的登录凭证。（如果想了解其默认值的话可以转到 4.2 节。）在登录进去的管理面板上，你会看到一些菜单选项，下面我们来介绍 3 个经常用到的菜单。



图 4-10

Gateways 菜单。Gateways（网关）菜单提供了添加 IP 地址及端口的配置（见图 4-11）。这些配置可使 ElectroServer 和任何与之相连接的客户端实现 Socket 通信。

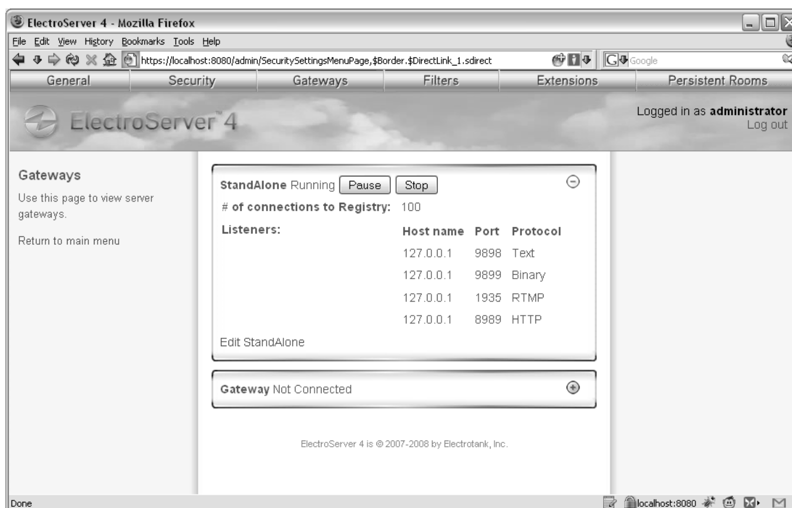


图 4-11

如果需要编辑这些地址中的任一项，点击 Edit StandAlone（单独编辑）按钮，它会提供这些监听器的配置（如图 4-12 所示）。

对于需要的 IP 地址和端口组合，只要在列表末尾添加新行即可。在本书范围内其默认值已经够用了。

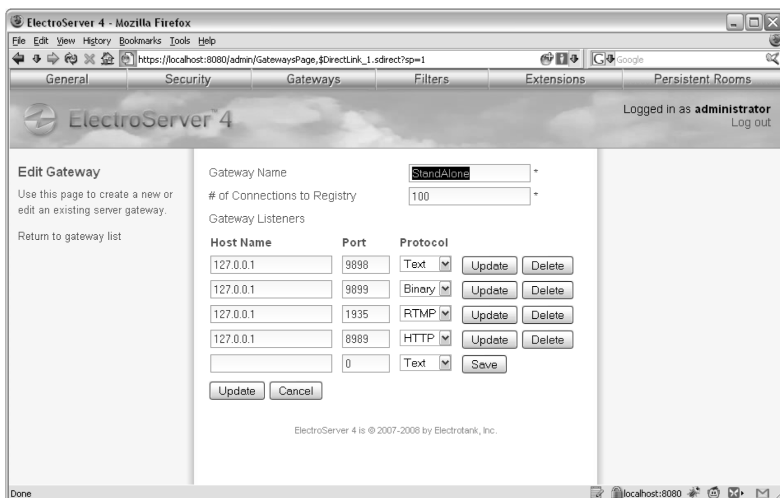


图　4-12



提示　你只能对在运行 ElectroServer 的服务器上可用的 IP 地址和端口组合创建监听器。如果地址在服务器上不可用，或者端口被占用，那么 ElectroServer 在下次重启后会报错。

Extensions 菜单。在 Extensions（扩展）菜单中，你能安装并配置任何将要被 ElectroServer 使用的服务器扩展。Extensions 菜单的当前页面会提供一个目前已安装的扩展的列表以及安装新扩展的选项（见图 4-13）。我们将在附录中详述扩展的安装。

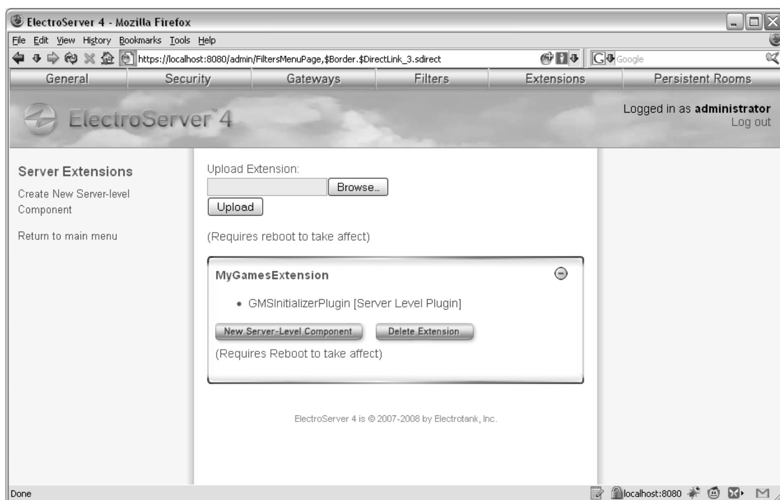


图　4-13

Persistent Rooms 菜单。这个菜单主要提供了创建在 ElectroServer 运行时（且不论何时）一直存在的房间和区的方法（图 4-14）。举例来说，它们对于持久存在的房间（比如游戏大厅或者人们经常会去玩的公众性游戏）来说是很有帮助的。

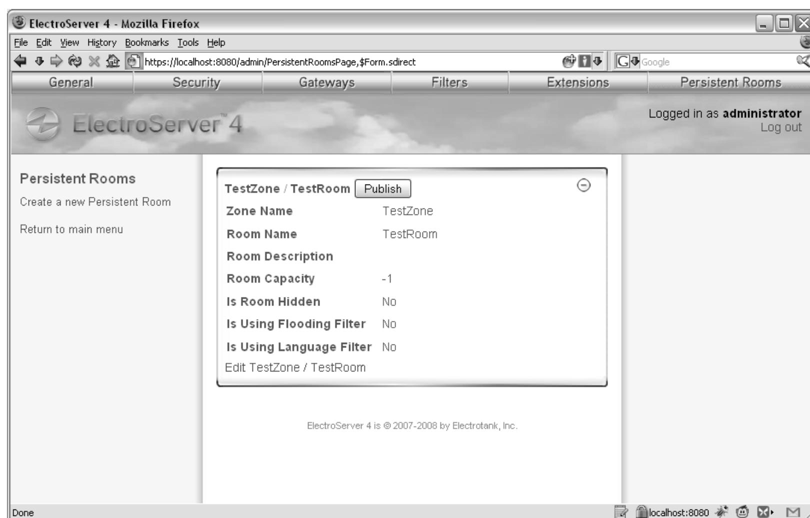


图 4-14

固有型房间要配置的选项和那些使用客户端 API 创建房间过程中需要配置的选项相同。它们不仅对于我们在任意客户端程序中实现聊天功能时会很有用，而且对于一经登录即自动加入的任一连接客户端依然有效。

第 5 章

聊　　天

所有的多人游戏与虚拟世界都有聊天功能。用户可以发送信息给其他人，对方可以立即看到该信息。用户可以用这种方式来了解彼此、插科打诨，或者在团队游戏中进行战略配合。

在本章中，我们来看看各种不同的用户聊天方式，以及这些方式如何与 ElectroServer 协同运作。我们将探讨一些新的房间概念，然后看看用于建立一个简单聊天室的源代码。

5.1　概述

要想设计一个聊天程序，你必须熟悉一些概念。这些概念会决定你在何处聊天、谁会看到你的聊天消息以及这些消息在一开始是如何被创建的。本节我们将介绍这些基本概念以及语言过滤问题。

5.1.1　聊天能见度

当一条聊天消息从用户发送到服务器端上时，谁会看到它呢？答案就取决于这条聊天消息的能见度。在大部分聊天应用程序中，聊天信息被设定为两种，不是公开消息就是私密消息。

公开聊天消息是指用户在知道某范围内的其他用户将会看到此消息的情况下发送给服务器的一种消息。一般情况下，公开聊天会在同一房间内的用户间发生。房间中的任何一个用户都可以直接往该房间发送公开聊天消息，然后此消息就会被广播给该房间中的所有用户（图 5-1）。

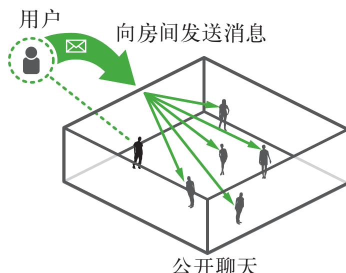


图　5-1

私密聊天消息（有时也叫做悄悄话）是指用户直接发送给一个或多个其他用户的一种消息（图 5-2）。私密消息可以发送给某个用户或者借由填充数组发送给多个用户，并且发送私密消息的用户可以不必在房间中。

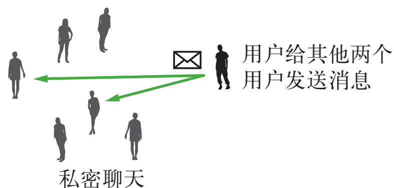


图 5-2

总的来说，以上所讲到的公开和私密的聊天形式在绝大部分多人网络游戏中都能找得到。不过，在你自己设计程序时，还可以通过其他方式来确定消息的能见度。例如，你可以编写一个“吼叫”（shout）的聊天消息——它可以被发送到每一个连接到服务器上的用户，或者只被发送给 30 级或更高级别的盗贼。

在一些要求更为严格的针对儿童开发的虚拟世界中，除非经家长许可，否则孩子们不会看到任何聊天消息。在那种情况下，即使这些消息存在，孩子们也看不到它们。家长可以让孩子们看到仅来自其伙伴的聊天消息、任何人发来的聊天消息，或者在其他自定义规则控制下可见的消息。

5.1.2 聊天类型

在大部分的多人游戏和虚拟世界中，你能发现的两种主要的聊天模式分别是自由进入式（图 5-3）和基于列表式。大部分儿童虚拟世界的聊天功能都结合了这两种模式，而这两种模式聊天功能的开启或者禁用则是由家长来控制的。

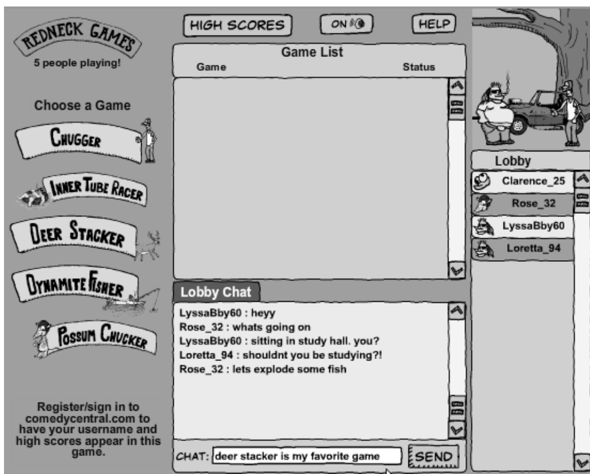


图 5-3

自由进入式聊天是人们所希望的标准聊天模式。用户可以自由地用键盘来输入他们想说的话。尽管在输入话语的时候可能会存在一些语言过滤，但在大部分情况下是不受约束的。

基于列表式聊天有时也被叫做*安全聊天*，它指的是让用户从一个分级列表中选择聊天词组来进行聊天（图 5-4）。尽管这种方式被普遍地使用且有些许优势（下面将讨论），但同时它也会令人沮丧。毋庸置疑，更多的聊天与用户交互行为都是发生在用户能够创建他们自己的聊天消息时。



图　5-4

基于列表式聊天有几个优点。首先对于孩子们来说，它被认为更加安全，因为它将严格约束孩子们能要说的话。另一个优点是它可以让不同语言的用户“直接地”交流。在源代码中，每一个词组都有一个唯一的 ID。当用户进入聊天程序时，基于列表的聊天数据是根据用户所惯用的语言而被加载进来的。当用户发送一个聊天词组，事实上他们发送的仅仅是一个与词组相关联的 ID。接受方客户端使用这个 ID 来决定显示哪一个词组（见图 5-5）。因此一个在阿拉巴马州正查看程序的用户发送的是“Hello”，而在巴塞罗那的用户将会看到“Hola”，同时另一个在巴黎的用户将会看到“Bonjour”，等等。

尽管自由进入式聊天与基于列表式聊天是两种主要的聊天模式，但却不是仅有的聊天模式。语音和视频聊天相对少见，但确实存在。此外，我曾经见过不止一个多人游戏让用户从一个已核准过的列表中通过拖放单词来组合成句子从而进行聊天。



图 5-5

5.1.3 房间概念

在第 4 章中我们曾经简要地讨论过房间，在本章中我们将继续关注房间，因为它们将更广泛地应用于 ElectroServer。

1. 快速回顾

简单地说 , 房间是用户的集合，大厅是房间的集合。房间概念的存在是为了提供一些管理效能，以便群体用户聚集与互动。一般情况下，在任何服务器解决方案中用户都可以同时加入一个或多个房间，然后通过聊天和那个房间（或那些房间）里的用户进行交互。

通过 ElectroServer，在一个房间中的任何用户都可以直接向该房间发送一条消息。这就是一条公开聊天消息，它可以广播给所有在那个房间内的用户。私密聊天消息可以从任一用户发送给任何其他用户，而不必考虑他们所在的房间。

正如第 4 章讨论过的一样，有两种形式的房间：动态型和固有型。只有被需要时，动态型房间才被临时地创建起来。用户可以使用 API 去创建一个房间然后加入进去。其他用户也可以加入到这个房间中。最后，当所有的用户都离开房间时，系统会自动将这个房间从服务器端移除。固有型房间是一直存在的。使用 ElectroServer 管理面板，可以配置固有型房间。固有型房间与动态型房间在每一方面都是相同的，除了一点，即当没有用户时固有型房间不会被移除。

2. 房间行为

用户通过使用客户端 API 加入一个房间的方法有两种。

- JoinRoomRequest —— 这个请求对象用于加入一个已经存在的房间。用户需指定房间以及房间所属区的 ID。
- CreateRoomRequest —— 这个请求用于创建或加入一个房间。在动态房间的设置中我们通常用这个请求来尝试加入一个房间，因为如果房间不存在的话它会创建出房间。

我们将在本章随后的聊天范例文件中用到这些请求对象。

用户可以在加入房间时通过请求对象创建用户特定设置，而且房间创建者在创建时也可以建立房间特定设置。这里有太多需要查看的设置，我们只看看通常用到的设置，其他的设置我们使用默认值。

以下是一些加入房间时我们需要指定的设置。

- 接收用户列表更新（receive user list updates）——默认值为 true。如果其值为 true，则不论何时，只要有人加入或者离开房间，服务器端都会把该情况通知用户。在不需要显示用户列表或者用户的加入与离开都很频繁时，把这个参数设置成 false 是合理的。
- 接收房间列表更新（receive room list updates）——默认值为 true。如果其值为 true，便会将用户所在房间所属区内其他房间告知用户。增加新房间、删除房间或者当房间的任何公共属性（比如说房间类型）被修改时，用户都将得到通知。在交流异常活跃的区中，这个设置通常是关闭的，因为只有这样才能保证客户端不会被泛滥的通知所淹没。

以下是创建房间所需指定的一些设置。

- 容量（capacity）——房间所能容纳用户的最大值。尝试加入满员房间的用户将会收到一条表明该房间满员的错误事件。该参数的默认值为 –1，表示不限制房间容量。
- 隐藏（hidden）——默认值为 false。如果该值为 true，这个房间将不会出现在区的房间列表上。
- 插件（plugin）——如第 4 章所述，插件是写在服务器端的代码，它可以被实例化且其作用域可以为服务器端或客户端。它们是大部分游戏和虚拟世界实现自定义行为时所采用的方式。当创建房间时，你需要指定一个或多个要被创建的插件并且限定其作用域为该房间。



注意　其他的用户和房间设置（这里没有详细讨论）包含与语音流和视频流有关的设置，以及接收用户和房间变量更新的设置。

5.1.4　聊天过滤

当用户能够聊天时，你很快就会发现参与其中的用户可以划分成两大类：一类是社交者，一类是骚扰者。社交者把交际作为聊天目的。他们会谈论任何话题，比如说游戏或者时事要闻，或者他们会去结识另一客户终端上的人。而骚扰者的唯一目的就是要激怒其他用户。骚扰者会通过在聊天室中亵渎和羞辱对方，或者利用他们在游戏中的行为来达到其目的。

令人遗憾的是，除了将聊天转变成基于列表式之外，没有办法完全使用户不滥用聊天功能。你能采取的措施就是设法将用户滥用系统的方式减到最少。聊天过滤是你所能做的一件主要事情。

聊天过滤（有时被称作语言过滤）是一种分析聊天消息并基于该分析而采取一些措施的行为。通常情况下我们会使用聊天过滤来发现亵渎词语，然后删除这条消息。（在一些聊天过滤中有一种替代性做法是，找到那些低级词语然后用一些非攻击性词语来代替。）

1. 过滤器类别

大部分聊天系统依靠两种截然不同的层次来进行过滤：*黑名单和白名单*。在黑名单方案中，你会得到一个内含不该出现在聊天消息中的词语的列表——它们一般是低级的或简短的攻击性词组。如果在聊天消息中发现了列表中的词组或单词，那就会采取某些措施。白名单过滤器刚好相反。在此方案中存在的列表包含了每一个被允许出现在聊天消息中的单词。如果聊天消息中的一个单词没有在白名单中，那就会采取某些预定措施。

ElectroServer 内建有过滤器，默认情况下使用一个小黑名单。通过管理工具，你可以增加单词列表，并定义列表是白名单还是黑名单。然后你就可以决定每个房间使用哪一种过滤器了。

开发一种有效的聊天过滤策略的关键在于使用联动运作的过滤器组合。考虑由一个白名单和一个黑名单组成的过滤器联动组合。其目的是允许正当词语通过，而禁止不良词语通过。你可以指定整部词典是白名单而把你想删除的词语指定为黑名单。

一些高级过滤系统所能做的不仅仅是许可或屏蔽特定词语。在 Electrotank 公司，我们致力于为客户端提供一种过滤系统，它能发现暗示恋童癖引诱儿童行为的某些单词或词组。在大多数的情况下，它们是一些在多数语境中完全无害的单词或词组。因此我们不想彻底地将其屏蔽。所以当发现这种类型的措辞时，我们将生成那个用户聊天副本的记录，它保存了该用户在此之前和随后几分钟内所说的所有话语。接着这个聊天副本会提交给审查员进行核查，然后他们会采取措施将这些消息屏蔽，或者有可能向官方举报该用户。



注意　有些公司不对用户的聊天话语加以限制，但得由一个人去审查每条聊天消息。我们有个顾客曾经想要这样一个系统。在这种情况下，用户输入一条消息并将其发送。这条消息最终会出现在审查员的电脑屏幕上。审查员将手工地批准或否决这条（以及所有的）聊天信息。这活儿可够累的！

2. 性能关注

在实际当中，聊天过滤工作最终可能会消耗掉可观的（有时是惊人的）资源总量——无论是从开发方面还是从原始计算能力方面来说。有些知名的虚拟世界仅在聊天过滤上就耗费了超过 50% 的计算资源。

典型的白名单或黑名单过滤器包含成百乃至上千个用来匹配的单词。由蛮力匹配转变成基于特里结构（Trie-based）的匹配方案会极大地节省执行时间。ElectroServer 自带的过滤器使用了特里结构。你可以去维基百科（http://en.wikipedia.org/wiki/Trie）了解更多有关特里结构的知识。

建立可扩展的过滤系统

尽管你在过滤系统的最优化上做出最大的努力，但它仍需要大量的计算能力。一种解决方案是彻底地建立无状态（stateless）且可分配的聊天过滤系统。这样就可以由多台服务器而不是仅由一台处理。这里所说的“无状态”指的是你可以在运行时把所有需要的信息传递给过滤器，因此它们就能极少或根本不需访问虚拟世界里的状态信息了。这会使你能够将过滤分发给相对低廉且独立的计算节点以获得灵活的可扩展性，而不会使核心虚拟世界的负载增加。

5.2　一个简单的聊天室

既然我们已经讨论了聊天室的一般概念以及它们是如何应用于 ElectroServer 的，那么就来看一个专为本章创建的简单的范例（图 5-6）。接下来我们会描述在这个范例中出现的功能，然后根据这些功能逐步讲解代码的相关部分。

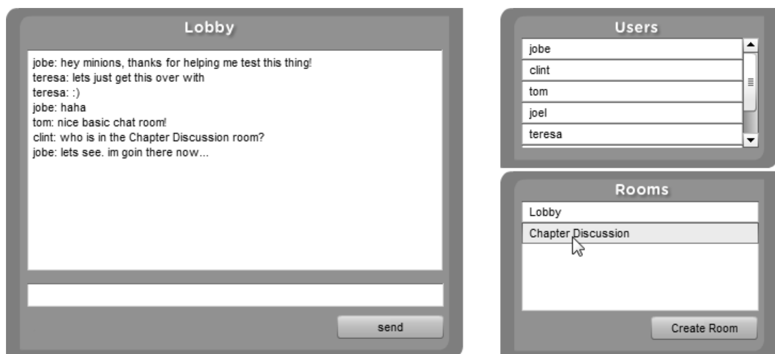


图　5-6

5.2.1　功能

这个简单的范例所允许用户实施的操作见诸典型的聊天室中。它们包括：

- 登录；
- 向房间发送聊天消息；
- 发送私密消息；
- 查看房间中的用户列表；
- 查看区中的房间列表；
- 加入房间列表中的任何一个房间；
- 建立新房间。



注意　尽管本例中编写的功能就只有这么多，但如果你想要一个功能更丰富的体验，你还是能够轻松地添加其他功能的。你可以试着在创建房间界面上添加更多选项或者添加语音 / 视频聊天功能。

5.2.2　逐步讲解

这个项目使用的 Flash Develop 项目文件名叫做 Chat Room.as3proj。如果你想在阅读本节时浏览源代码，可以打开该项目文件并启动 ElectroServer。



注意　在 book_files/chapter 5/chat_room 可以找到此例的源代码。

下面介绍大体的程序流程。当聊天程序被启动时，它加载了一个 XML 文件用来告知它要连接的 IP 地址和端口号。然后聊天程序会试着用那些设置去连接 ElectroServer，并显示给用户一个标识此连接过程的界面。如果连接成功，则聊天程序将去除连接界面并为用户显示登录界面。当然在此界面中用户可以输入并提交用户名。然后登录界面立刻被移除，同时把登录请求发送给服务器端。然后服务器端会发送一个登录响应给客户端。如果登录成功，用户将看到聊天室用户界面并加入到一个默认房间中。

在该聊天室界面上，用户可以查看用户列表、房间列表以及聊天信息。用户不仅可以发送公开和私密消息，同样他们也可以创建新房间或者加入到房间列表下的任意房间中。

1. 配置文件

如同本书所有范例一样，源代码都编译进 bin 目录中。如果查看 bin 目录，你就能看到一个叫做 server.xml 的文件。这个 XML 文件包含服务器连接设置（IP 地址和端口号）。聊天程序加载这个文件，提取连接设置并使用它们去连接 ElectroServer。

```
<server>
  <connection ip="127.0.0.1" port="9899" />
</server>
```

像这样把连接设置放在程序外部的做法可以使你不用重新编译聊天程序就能改变它所使用的 IP 地址和端口号。

2. 聊天流程类

这个类用来加载 XML 文件并连接 ElectroServer，以及管理用户登录。在第 4 章中，我们了解到如何去创建一个 ElectroServer 类的新实例以及如何建立一个连接然后登录，因此我们不必在此重复了。不过我们要谈一个在这里会用到的新事件 `ConnectionClosedEvent`。

如你所知，客户端和服务器端会建立一个 Socket 连接。如果该连接因任何原因而被切断，

那么就会触发 ConnectionClosedEvent。连接丢失可能是由于你的网络接入点与服务器端之间出现了一些问题，可能是服务器被关闭或者服务器强制断开连接，也可能是客户端故意断开连接。不管是因为何种原因，能捕获到一个能告知你连接已关闭的事件毕竟是件好事，这样你就能通知用户或者建立一个新连接。

在 initialize 函数中，我们为这个事件加入了一个事件监听器：

```
_es.addEventListener(MessageType.ConnectionClosedEvent,
➔ "onConnectionClosed", this);
```

接下来你就能在这个类中发现 onConnectionClosed 函数：

```
public function onConnectionClosed(e:ConnectionClosedEvent):
➔ void {
  showError("Connection was closed");
}
```

本例中，我们只是通知用户连接是否丢失。你可以通过先连接并随后关闭 ElectroServer 来对此进行测试。showError 函数只是简单地为用户提供一个内含消息的警告显示（见图 5-7）。当初始连接尝试失败或登录尝试失败时，也将使用该函数。

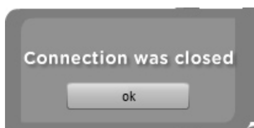


图 5-7

3. 聊天室类

在用户成功登录后，我们将创建该类的一个实例并传入一个 ElectroServer API 实例。这个类处理所有的客户端与服务器端的通信以及聊天室用户界面。

当我们创建了该类的实例并且在其中设置了 ElectroServer API 的实例后，initialize 方法就会被调用。

```
public function initialize():void {
  //add ElectroServer listeners
  _es.addEventListener(MessageType.JoinRoomEvent,
  ➔ "onJoinRoomEvent", this);
  _es.addEventListener(MessageType.PublicMessageEvent,
  ➔ "onPublicMessageEvent", this);
  _es.addEventListener(MessageType.PrivateMessageEvent,
  ➔ "onPrivateMessageEvent", this);
  _es.addEventListener(MessageType.UserListUpdateEvent,
  ➔ "onUserListUpdatedEvent", this);
  _es.addEventListener(MessageType.ZoneUpdateEvent,
  ➔ "onZoneUpdateEvent", this);
```

```
  //build UI elements
  buildUIElements();

  //join a default room
  joinRoom("Lobby");
}
```

这个函数为服务器端事件注册了多个事件监听器，并调用了一个函数来建立用户界面，然后加入了一个默认的聊天室。添加事件侦听器用于处理接收聊天消息、更新用户列表和房间列表，并且当你已成功地加入到一个房间时它能获知此消息。在本节余下内容中我们将查看每一个事件处理器以及一些新的请求对象。

4. CreateRoomRequest 事件与 LeaveRoomRequest 事件

joinRoom 函数用来将用户加入到其指定的房间（图 5-8）。这个函数引进了两种新请求对象 CreateRoomRequest 和 LeaveRoomRequest。

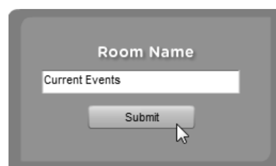


图 5-8

CreateRoomRequest 对象的默认服务器端行为是创建用户指定的房间，或者如果该房间已存在的话，它会将用户加入到该房间。如果该房间已经存在，此对象会被配置成返回一个失败（failure）。此处将涉及很多 CreateRoomRequest 对象所能用到的附加与高级设置。但默认设置对于大多数应用都很有效。

```
//create the request
var crr:CreateRoomRequest = new CreateRoomRequest();
crr.setRoomName(roomName);
crr.setZoneName("chat");

//send it
_es.send(crr);
```

你刚才看到的代码是出现在 joinRoom 函数中的。首先，我们要建立一个请求对象，然后根据 roomName 函数参数在该对象中设置目标房间名称。我们还将指定这个要创建的房间所属的目标区（此区的名字可以任取）。配置好该对象之后，我们就将它发送给 ElectroServer。如果用户成功地加入到该房间，那么接下来就会调用 onJoinRoomEvent 函数。我们稍后再来讲这个函数。

下面这个条件语句也出现在 joinRoom 函数中：

```
if (_room != null) {
  //create the request
  var lrr:LeaveRoomRequest = new LeaveRoomRequest();
  lrr.setRoomId(_room.getRoomId());
  lrr.setZoneId(_room.getZoneId());

  //send it
  _es.send(lrr);
}
```

稍后，当查看 onJoinRoomEvent 函数时，你会看到我们存储了一个关于 Room 对象的引用，你可以把这个引用归属为一个叫 _room 的类属性。在上面的代码中，如果 _room 不为 null,那么就意味着你在一个房间中，那么我们可以发送 LeaveRoomRequest 使你离开该房间。ElectroServer 允许你一次加入多个房间，但是既然用户界面只允许你和一个房间发生互动，那么我们就确保你每次只在一个房间中。在上面代码中，我们创建了一个新的 LeaveRoomRequest 对象，并且在其中设置了房间 ID 和区 ID。然后它就会被发送给 ElectroServer。

5. **UserManager** 类与 **ZoneManager** 类

除了为客户端和服务器端间的通信提供框架以外，ElectroServer API 还记录下了你所在的所有区内的房间以及你所在的所有房间中的用户。这样就非常方便，因为你不必通过监视所有添加和移除的项目来亲自管理列表了。但如果真想那么做的话，你就必须知道相应的细节。

在此应该介绍下面 3 个类类型。

- 用户——该类的实例用于描述在服务器端上的实际用户。UserManager 类维持着一个含有当前你（客户端）看到的全部用户的总控列表。
- 房间——该类用于描述区中的房间。你所处房间的 room 对象包含着一个该房间的用户列表。
- 区——该类用于描述你所在的区，它含有你能在该区看到的房间的列表。ZoneManager 类维持着一个含有你当前所在的所有区的总控列表。

你会在稍后的应用中看到它们。

6. **JoinRoomEvent** 事件

当用户加入房间后，onJoinRoomEvent 函数就会被调用（这并不奇怪）。在该范例代码中，我们使用 JoinRoomEvent 事件来告知我们正处于一个新房间中，接着，作为结果我们将更新用户界面。

```
public function onJoinRoomEvent(e:JoinRoomEvent):void {
  //the room you joined
  _room = e.room;
```

```
  //update the display to say the name of the room
  _chatRoomLabel.label_txt.text = e.room.getRoomName();

  //refresh the lists
  refreshUserList();
  refreshRoomList();
}
```

JoinRoomEvent 对象包含了一个对你刚加入的房间的引用。我们把该引用存储到类属性 _room 中。接着我们就会更新用户界面以便在历史记录字段中显示聊天室名称。

最后，该函数所要做的就是刷新用户列表与房间列表。下面我们就来看看这是如何做到的。

7. 刷新用户列表与房间列表

用户列表与房间列表使用的都是标准的 Flash List 组件（使用这个组件我们就不必自己重写了）。我们为该组件提供了一个用于填充其内容的 DataProvider 类的实例。在用来填充用户列表与房间列表的函数中，我们创建了一个新的 DataProvider 实例并为其添加数据，而后把它设置到列表组件中。

下面是用来刷新用户列表的函数：

```
private function refreshUserList():void {
  //get the user list
  var users:Array = _room.getUsers();

  //create a new data provider for the list component
  var dp:DataProvider = new DataProvider();

  //loop through the user list and add each user to the data
  ➔ provider
  for (var i:int = 0; i < users.length;++i) {
       var user:User = users[i];
       dp.addItem( { label:user.getUserName(), data:user} );
  }

  //tell the component to use this data
  _userList.dataProvider = dp;
}
```

首先我们从房间对象中提取出用户列表数组。这是一个 User 类实例数组，数组中每个元素都代表着房间中的一个用户。然后创建了一个新的 DataProvider 对象。而后我们遍历用户列表数组并为这个数据提供程序加入每个用户的格式化对象。这种格式化对象利用一个 label 属性（一个字符串）来确定列表中要显示的项目名称，它还利用一个 data 属性来确定与列表项目相关联的数据。最后，我们将该数据提供程序设置于组件中以使组件显示数据。

现在看看用来刷新房间列表的函数：

```
private function refreshRoomList():void {
  //get the zone
  var zone:Zone = _es.getZoneManager().getZoneById
  ➔ (_room.getZoneId());

  //get the room list
  var rooms:Array = zone.getRooms();

  //create a new data provider for the list component
  var dp:DataProvider = new DataProvider();

  //loop through the room list and add each room to the data
   ➔ provider
  for (var i:int = 0; i < rooms.length;++i) {
       var room:Room = rooms[i];
       dp.addItem( { label:room.getRoomName(), data:room} );
   }

   //tell the component to use this data
   _roomList.dataProvider = dp;
}
```

该函数与 `refreshUserList` 函数遵循相似模式。它建立了一个 `DataProvider` 对象并将其设置于房间列表组件中以使数据得以显示。这里有一些你以前没有接触过的内容：如何访问 `ZoneManager` 并从其中提取出一个区，然后再从该区中获取房间列表。该函数首先创建了一个你所在区的本地引用。这个区对象包含了一个在该区中所有房间的列表，该列表能够被 `zone.getRooms()` 数组所访问。该数组中的每一个元素都是一个房间对象，`DataProvider` 对象利用它们来生成一个房间列表。

8. 发送聊天消息

点击 Send 按钮就会执行 `onSendClick` 函数。这个函数包含着一种逻辑用来检查消息字段以确保其内容不为空白，并且当其不为空白时，该逻辑将分析其中内容以确定待发送的消息到底是公开消息还是私密消息。

可以发送公开或私密消息是这种聊天的一个特点。如果在消息字段内正常地输入内容然后点击 Send 按钮，你就会发送一条公开消息。如果你键入一个正斜杠（/），其后跟着一个用户的名字，然后再键入一个冒号（：），那么消息的其余部分将做为一条私密消息发送给那个用户（图 5-9）。例如，下面的消息字段内容会将“due, omg!”发送给 Fezik。

```
/Fezik:dude,omg!
```

看看 `onSendClick` 函数中的这行代码：

```
if(msg.charAt(0) == "/" && msg.indexOf(":") != -1){
```

msg 变量是一种包含了消息字段内容的字符串。如果该字符串第一个字符是正斜杠并且随后在其中再出现冒号的话，那么你即可认定发送的是私密消息，程序接着就会执行上面的 if 语句中的分支语句。

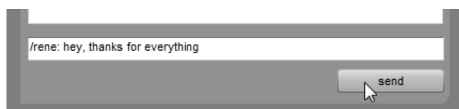


图　5-9



注意　在第 4 章中我们已经查看过 PrivateMessageRequest 请求了，因此在这里就不再讨论那些代码了。PublicMessageRequest 请求对象处理的方式与之相同——传送消息到用户名数组中。在本例中，数组中仅有一个用户名。

如果已确定消息是公开消息，那么我们将使用 PublicMessageRequest 请求对象。下面就是创建和发送该请求对象的代码：

```
//create the request object
var pmr:PublicMessageRequest = new PublicMessageRequest();
pmr.setMessage(_message.text);
pmr.setRoomId(_room.getRoomId());
pmr.setZoneId(_room.getZoneId());

//send it
_es.send(pmr);
```

首先创建一个新的 PublicMessageRequest 请求对象，接着我们在其中设置聊天消息文本。因为你可以同时在多个房间内，所以你需要指定要将消息发送到哪个房间，因此还要在该请求对象中设置房间 ID 和区 ID。然后往服务器端发送该请求对象。

私密与公开聊天消息请求对象都会触发事件。下面我们来看看这一点。

9. 接收聊天信息

这部分简单！接受到一条聊天消息后将触发 onPublicMessageEvent 或者 onPrivateMessageEvent。在这两种情况下，我们都使用在事件对象上的数据来为历史字段增加聊天项目：

```
public function onPublicMessageEvent(e:PublicMessageEvent):
➔void {
  //add a chat message to the history field
  _history.appendText(e.getUserName() + ": " + e.getMessage() +
  ➔"\n");
}
```

在这个函数中，我们提取出向房间发送消息的用户的名称以及他们所发的消息内容，接着我们会把这些都添加到聊天消息历史字段中。

下面来处理私密消息：

```
public function onPrivateMessageEvent(e:PrivateMessageEvent):
➔ void {
  //add a chat message to the history field
  _history.appendText("[private] "+e.getUserName() + ": " +
  ➔ e.getMessage() + "\n");
}
```

处理私密消息与处理公开消息的方式相同，所不同的是我们在私密消息前面加入了一个`[private]`字符串，简单吧？

第 6 章

游戏逻辑决策位置

在多人游戏与虚拟世界中，客户端与服务器端都存在着游戏逻辑。但如果游戏逻辑在两处都存在，那么哪一方有权做出重要逻辑决策？这样做的理由又何在呢？本章将回答这些疑问。选择开发时能维持游戏状态与放置游戏逻辑代码的最佳位置，可以缩短你后续查找 bug 的时间并最终使游戏更为安全。

我们先在本章的起始部分介绍“权威”（负责人）这个概念，余下部分将会介绍本书中所用到的第一个服务器端插件以及一些新的 ElectroServer API 知识，然后我们会为你逐步讲解一个简单的 ActionScript 范例游戏的制作过程。

6.1 一些新概念

如果你习惯于编写单人游戏，那么你肯定也习惯于把所有游戏逻辑代码放进 Flash 客户端中，那样也就不必考虑在何处裁决游戏逻辑了。而在多人游戏中则需要考虑在哪个位置裁决游戏逻辑。我们把多数重要的游戏逻辑是在客户端上裁决的游戏类型称之为*客户端权威型*（authoritative client），而把多数重要的游戏逻辑是在服务器端上裁决的游戏类型称之为*服务器端权威型*（authoritative server)。

6.1.1 客户端权威型

让我们来看几个客户端权威型的游戏范例。在这些范例中，服务器端只被当作是客户端间传递消息的一种渠道。我们会介绍一个客户端上成功地进行逻辑裁决的范例和一个失败的范例。

我们先来考虑一个双人回合制的桌球游戏（如图 6-1 所示）。

在这个游戏中，所有游戏逻辑都包含在 Flash 客户端中。游戏开始时玩家双方在各自的客户端上都能看到完全相同的球势布局。当轮到某一玩家击球时，他能使用球杆去选择击球角度与力度（击球角度与力度如图 6-1 所示）。一旦该玩家按下鼠标击打母球，立刻就会有一条包含击球角度值与力度值的消息从玩家 A 发送到玩家 B。双方客户端因为接受了相同的输入内容（击球角度与力度）故而能够一致地将游戏玩下去。双方玩家总能知道桌球何时应该落袋、回合何时结束以及哪一方是赢家。客户端权威型在这个范例中效果很好。

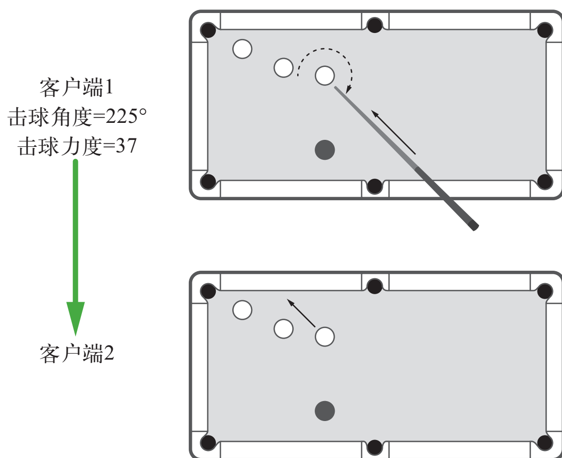


图 6-1　客户端权威型：对于简单的桌球游戏来说它是不错的选择

现在让我们再来考虑一个双人实时坦克射击游戏的范例（图 6-2）。在这个游戏中，用户可以利用键盘输入来迅捷地操纵坦克。无论炮塔的朝向如何，只要一点击鼠标，坦克就会开火。炮弹从炮塔中射出并沿直线行进。

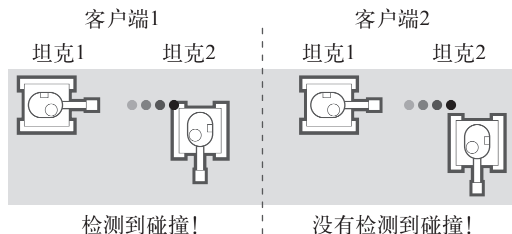


图 6-2　客户端权威型：对于实时坦克射击游戏来说它不是一个良好的选择

不要自作聪明

如果你是一个聪明的程序员并且想设法避免编写自定义服务器端代码，那么你可能会想一些备选方案来解决在上例坦克射击游戏中所暴露出来的玩家判断不统一的问题。比如说你会考虑一种方法使得一旦检测到碰撞发生，玩家彼此间就可以发送消息通知对方。只有在双方玩家都同意的情况下才能让客户端对碰撞做出反应。这就只有一种判断结果了。

但我得建议你避免使用这种方法。它不仅会因为双方客户端需要同步化而导致反应延时，而且它还极难调试，并且代码记录会变得超级复杂。更不要说两客户端其一可能会作弊以使其永不承认己方坦克被击中，从而使它变得无敌！

让我们看看这个游戏中的情形。假如当时坦克 1 是面向坦克 2 的，并且两者都没有运动。客户端 1（坦克 1 的控制方）点击鼠标向坦克 2 开火。炮弹离开炮塔飞向坦克 2。客户端 2 看着炮弹向自己的坦克飞来，因此在最后一刻开始开动他的坦克以避免它被摧毁。我们所陷入的困境是：客户端 1 会检测到有碰撞发生而客户端 2 没有检测到。客户端权威型方法在本例中不起作用。

6.1.2　服务器端权威型

我们在上面的范例中看到客户端权威型是如何导致客户端间判断产生分歧的。对于此种问题而言，解决办法是把重要的游戏逻辑决策转移到服务器端。在一个服务器端权威型架构中，服务器端就是唯一的游戏逻辑决策者，因此不会存在任何不确定的进程行为。

当把这种方法应用到上述坦克游戏范例中时，服务器端就会跟踪整个游戏的状态。因为服务器端对所有坦克和炮弹的位置都了如指掌，所以它就能在碰撞发生时进行游戏逻辑决策并把结果告知客户端（见图 6-3）。

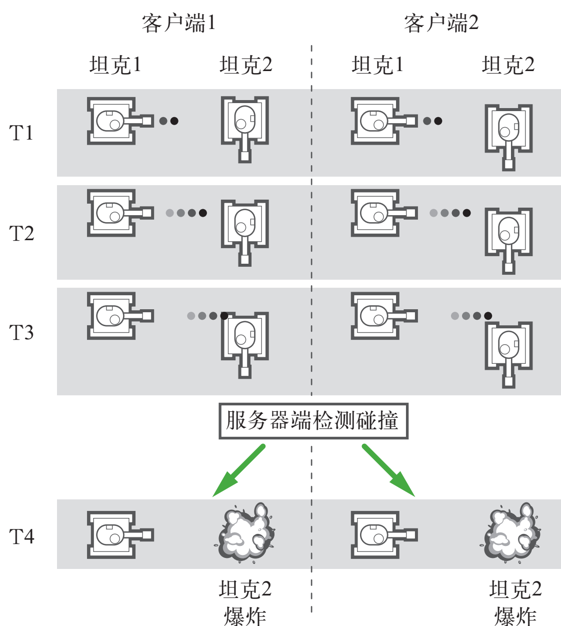


图 6-3　与上例同样的实时坦克射击游戏，所不同的是现在由服务器端来控制游戏逻辑决策

这里你会看到在我们上述射击游戏情形中的 4 个时间阶段内客户端所呈现的不同形态。你能发现在前 3 个时间阶段中，在客户端 1 看来，炮弹好像最终击中了目标，但是从客户端 2 的角度上看，炮弹却稍微偏离了目标而没有击中。不过最后由于服务器端判断炮弹击中了目标，所以双方客户端也都将该结果记录为“碰撞”。很显然客户端 1 是优胜者！

6.1.3　何时采用何种模式

你如何知道何时应该用一种方法而不用另一种呢？尽管对于这种问题而言并没有什么万灵解药，但是存在于某些类型游戏中的一些趋向性的东西以及一些自己提出的问题却能导引你到正确的选择方案上去。

所有信息都是已知的回合制游戏最有可能适合采用客户端权威型模式，我所说的“所有信息都是已知”意指信息处理不存在偶然性与隐含性。符合这些标准的游戏包括国际象棋、跳棋、四子棋还有落袋桌球。你能轻松地创建任何这种类型的游戏并且在客户端运行游戏逻辑代码。

多数实时类游戏（不存在回合的游戏）自动应归属于服务器端权威型游戏。另外，在有些信息处理上存在着偶然性或者尚未透明的某些回合制游戏也适宜用服务器端权威型来处理。在这里我将介绍几种游戏并且告诉你为什么它们应该是客户端权威型或服务器端权威型。

- 德州扑克（Texas Holdem Poker）——这种游戏要求一副洗好的牌与对家的牌都不能为人所知。如果在客户端上维持像这样的游戏逻辑，那么聪明人就可以利用内存读取工具来查看洗好的牌或者通过创建一个被恶意修改过的客户端从而洞悉他们对家的牌。如果在网络游戏中下注用的钱是现实货币，则必须由服务器端裁决游戏逻辑。
- 地产大亨（Monopoly）——它是绝大多数现代桌面式游戏的良好样板。骰子一掷，机会现前，两副牌秘而不宣。该游戏理应由庄家裁判！
- 赛车类游戏——比赛中最先冲过终点线的人是赢家。你不能指望所有客户端都精确地认同比赛中全部车辆所处的位置，尤其当车速非常快的时候。正因为如此，服务器端必须得记录车辆的方位，它才是最终的游戏逻辑决策者。

可能你会感到困惑，现在在有些游戏中可能既存在客户端权威型部分又存在服务器端权威型部分。通常在这种混合式处理方法中，客户端权威型的作用无关轻重，拿德州扑克游戏来说，你可以编写功能来嘲弄对手——咕哝着羞辱对手、摇晃桌子、晃动钥匙弄得叮当乱响。这些功能都不会直接影响游戏结果，所以它们可以完全由客户端进行操控。



提示　如存在疑问，则最好把游戏逻辑交付服务器端裁决。

6.2　ElectroServer 插件概念

我们在第 4 章中接触到了“服务器端扩展”（server extension）这个概念。“扩展”被用来为服务器端提供额外功能。共有 3 种类型的扩展：托管对象（managed object）、事件处理器（event handler）以及插件（plugin）。本节将温习插件的概念，然后将学习如何从客户端调用它们。

6.2.1 插件

插件就是运行于服务器端的自定义代码。如果要开发服务器端权威型游戏，你就必须得写插件。



注意 插件可以用 Java 或 ActionScript 来编写，在本书中使用的所有插件范例都是用 Java 编写的。

插件可以被概念化地理解为一个类。服务器端可创建出插件的新对象，后者的作用域为服务器端或一个房间。

既然游戏发生在房间内的集群用户间，那么相对我们的目的而言，最有用的插件是房间级插件（与服务器级插件截然相反）。在 ElectroServer 上使用的插件必须要先被安装（可查阅 6.3 节以了解个中关键）。你可通过获知扩展名称（extension name）及插件句柄（plugin handle）来指定某个安装好的插件。而当你使用 ActionScript API 来创建新房间时，也可以通过扩展名称和插件句柄来指定你想要创建的一个或多个插件并使其作用域为这个新房间。

6.2.2 与插件对话

创建好房间后，客户端就可以通过 API 与插件通信了。通过使用 EsObject 对象即可在客户端与插件间相互传递自定义信息。你可以用任何你需要的数据对 EsObject 对象进行格式化。



提示 在第 4 章中我们介绍了 EsObject 对象。因为本书的余下章节要广泛地使用它们，所以你可能得需要再仔细看看 4.1.6 节。

默认情况下 EsObject 对象是不包含任何数据的。因此如果想在客户端与插件间交换任何有用的信息，就必须往 EsObject 对象内添加自定义数据。我们将在本章稍后完整的插件与客户端通信中查看这些代码。

能够保证消息次序

客户端与服务器端通过使用 Flash 里现成的 Socket 类从而利用 socket 进行通信。Socket 类所使用的 TCP 协议能够保证每个发送出的消息都会抵达其目的地。如果消息在传输途中出现了一些问题，它就会被自动重新发送。需要着重指出的是，服务器端与客户端处理消息的次序与发送消息的次序相同。因此你可以放心大胆地连续快速发送给服务器端数个消息，它们会以同样的次序被服务器端接受并处理。

6.2.3 EsObject对象格式化方法

我找到了一个基本的 EsObject 对象格式化方法，它对于我最近几年所编写的每个多人游戏来说都非常有效。本书中所有范例都将使用这种方法，所以我们要在此讲讲它。

游戏中的每个在客户端与插件间传递的消息都可以被认为是一个行为（action）。所以我总是会确保每个用于客户端与插件间通信的 EsObject 对象都包含一个用于描述一个行为的变量。EsObject 对象剩余部分的格式化当然会取决于该行为的内容。

例如，让我们来看看回合制桌球游戏吧。客户端从来都只能处理一种行为——他们能够击球。从客户端发往插件的 EsObject 对象可能会包含这些变量：

- action = 击球；
- angle = 击球角度；
- power = 击球力度。

跟踪EsObject对象

EsObject 对象在游戏开发中发挥着很大的作用。客户端与服务器端开发者们需要统一 EsObject 对象格式以使这两端能够有效并准确地进行通信。但正如你在开发程序中所能预见的那样，在确定两端的 EsObject 对象格式是否统一的进程中经常会有 bug。通过在 EsObject 对象被发送前检查其中的内容，我们便可以轻松地调试部分代码。换句话说，你得跟踪 EsObject 对象。

例如：

```
var esob:EsObject = new EsObject();
esob.setString("name", "Magilicuddy");
esob.setInteger("age", 25);
trace(esob);
```

上述代码的跟踪输出结果应像这样：

```
{EsObject:
        name:string = Magilicuddy
        age:integer = 25
        }
```

不管 EsObject 对象的数据结构有多么复杂，你只需要通过跟踪就可以查看它的全部内容。不管对于服务器端还是客户端都是如此。当需要检视那些被发送或待接受的内容时，你和你的服务器端开发者会发现这个方法非常有效。

在桌球游戏中插件可以发送数个行为中的一个给客户端，比如像轮换击球方、击球、自落（母球被击落袋）以及游戏结束。“游戏结束”所对应的 EsObject 对象会包含这些变量：

- action = 游戏结束；
- winner = 赢家名称。

这个概念非常简单，但我还是通过开发了好几个游戏之后才认识到它的普遍性——对每个游戏中的任一行为它都有效。

6.3 安装扩展

游戏（包括本书中的所有范例）的自定义服务器端代码存在于扩展中。扩展必须位置明确以便于 ElectroServer 使用。你只需将扩展复制到 ElectroServer 安装目录里的 server/extensions 文件夹下即可。

以你从这里下载的范例文件为例，如果 ElectroServer 被安装在 C:\Program Files\ElectroServer4_0_6 文件夹下，那么当你复制了 GameBook 文件夹后，你就能在 C:\Program Files\ElectroServer4_0_6\server\extensions\GameBook 文件夹下找到 Extension.xml 文件。在将 GameBook 文件夹复制到正确的位置之后，重启 ElectroServer。



注意 你可以在 book_files/examples_extension/extension/GameBook 目录下找到这个扩展范例，如想对安装扩展了解更多，请参阅附录。

6.3.1 服务器级组件

如果扩展含有任何服务器级组件，你就得在它们首次被添加时使用管理面板将其标注为服务器级。每个服务器级组件只需标注一次。虽然此章我们本不需要此步骤，但是 GameBook 扩展所提供的 TimeStampPlugin 组件在后续章节中却需要被用作服务器级组件，所以我们现在就这么做吧。

重启 ElectroServer 之后，打开管理面板。如果你是在本地运行 ElectroServer，你会经常需要打开网页 https://localhost:8080/admin 来启动管理面板。以管理员身份登录之后，打开“Extensions”标签，在其上你会看到所有安装到 ElectroServer 上的扩展。点击 GameBook 旁边的加号按钮（+），再点击“New Server-Level Component”（新服务器级组件）按钮。下一个界面将在“Extension Name”（扩展名称）栏中显示出“GameBook”。从“Plugin Handle”（插件句柄）下拉菜单中选择“TimeStampPlugin”，而后在“Plugin Name”（插件名称）栏中输入 TimeStampPlugin（图 6-4）。

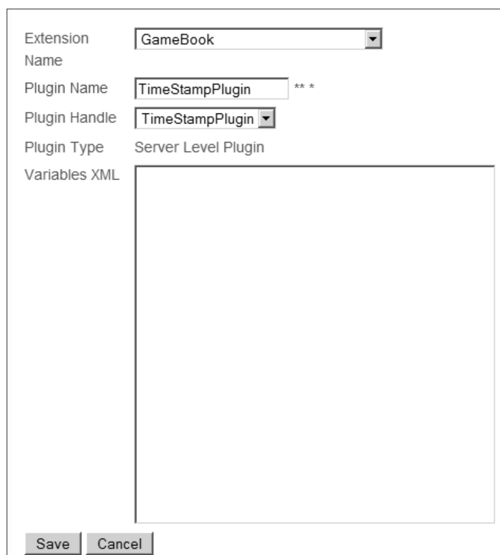


图　6-4

在添加完所有你所需的服务器级组件之后，再次重启 ElectroServer。请注意在这里你也会看到列出的其他插件。但此时不用管它们。



注意　插件句柄与扩展名称一起用于指定一个已安装的插件。插件名称是分配给该插件的一个实例的 ID。

尽管可以使用一个与插件句柄不同名的插件名称，但当两者相匹配时最易记录。

6.3.2　创建扩展

如果你要从头开始创建扩展，那么就需要以下必要的组成部分：

- 一个位于目录 server/extensions 中的正确命名的文件夹；
- 一个 extension.xml 文件，其中的 XML 节点（XML Node）指定了每一个托管对象、事件处理器以及用在该扩展中的插件；
- 一个含有运行 Java 代码所需的所有 jar 文件的 lib 文件夹，其中包括一个运行你自定义代码的 jar 文件。



注意　虽然有其他选项，但多数扩展的创建只需这些内容即可。可查阅附录来了解如何更容易地编译你的 Java 类、如何创建扩展并部署它，以及所有这些是如何在 NetBeans 中一步完成的。

6.4 挖宝游戏

至此，本章已经讨论了客户端权威型与服务器端权威型的概念，以及一些跟插件有关的ElectroServer新概念。现在让我们用一个挖宝游戏（图6-5）来了解一下这些概念的实际应用。



图 6-5

6.4.1 游戏特点

本游戏中玩家的鼠标指针被替换为小铲子的形状，背景画面是一片泥地。用户可以用他们的铲子在任意处点击以便在该点挖掘。在挖掘时先是会有一段2秒钟的铲子挖掘动画，然后会显示找到的物品或是显示空无一物。一旦某处已被挖过则不能再挖。如果找到物品就会得到分值奖励，你的总分值会在用户列表中更新。



注意 这个范例游戏没有结局。你只需运行这个SWF文件就能加入游戏，玩家可以随时加入或离开。

这个挖宝游戏是一个服务器端权威型游戏。客户端选择好了要挖的位置后将挖掘请求发送给插件。接着服务器端会检查该点是否被挖过，如果是的话则告知客户端不能再在那里挖掘；如果没有被挖过，那么服务器端会在等待2秒钟后告知客户端挖掘的结果：挖到物品或者什么也没有挖到。你可以挖到4种物品，它们的稀有度与分值各不相同。

6.4.2 逐步讲解

这里用到的Flash Develop项目文件为Dig Game.as3proj。你可以打开此文件以便在学习本

节时浏览源代码。如要测试该程序，你必须开启 ElectroServer 并保证安装了插件。



注意　在 book_files/chapter6/dig_game 可找到此范例的源代码。

因为该游戏只有一个界面所以其程序流程非常简单。游戏启动时加载了一个 XML 文件用于告知游戏要连接的服务器端 IP 地址与端口号。接着游戏会利用这些设置来尝试连接 ElectroServer。如果连接成功，服务器端就会以随机用户名使用户登录。待用户成功登录后，一个 DigGame 类的新实例就创建好了。

6.4.3　最小化交换数据

客户端与插件间使用自定义格式 EsObject 对象来进行通信。你可采用任意你想用的方式来格式化 EsObject 对象。但如果你希望传输大量的消息（由于游戏很受欢迎或者游戏自身需要的原因），那么最小化交换数据就会显得很有必要。关于这方面，PluginConstants 类会很有用。

PluginConstants 类只存储了被赋予了有用的名称与微小的数值的公共静态变量。下面列出本例中 PluginConstants 类里的所有变量：

```
// actions
public static const ACTION:String = "a";
public static const DIG_HERE:String = "d";
public static const DONE_DIGGING:String = "dd";
public static const INIT_ME:String = "i";
public static const PLAYER_LIST:String = "ul";
public static const ADD_PLAYER:String = "au";
public static const REMOVE_PLAYER:String = "ru";
public static const ERROR:String = "err";

// parameters
public static const ITEM_FOUND:String = "f";
public static const ITEM_ID:String = "id";
public static const NAME:String = "n";
public static const SCORE:String = "s";
public static const X:String = "x";
public static const Y:String = "y";

//errors
public static const SPOT_ALREADY_DUG:String = "SpotAlreadyDug";
```

服务器端插件包含着一个相似的类用于匹配以上所有值。我们会在另一个类中像这样格式化 EsObject 对象：

```
var esob:EsObject = new EsObject();
esob.setString(PluginConstants.ACTION, PluginConstants.
➔DIG_HERE);
```

```
esob.setInteger(PluginConstants.X, mouseX);
esob.setInteger(PluginConstants.Y, mouseY);
```

上述代码等同于：

```
var esob:EsObject = new EsObject();
esob.setString("a", "d");
esob.setInteger("X", mouseX);
esob.setInteger("Y", mouseY);
```



注意 相对于使用字符串 "a" 而言，使用 PluginConstants.ACTION 的优势在于编译时的检查及编程时的代码提示。如果编写代码时你打错了 PluginConstants.ACTION，立刻就会得到提示。而如果你敲错了一个字符串，除非发生运行时错误，否则你不会发现这个错误。

6.4.4 维持用户列表

既然我们准备加入到一个房间中，你可能会设想我们会使用 ElectroServer 内建的用户列表以及用户列表更新来为客户端维持用户列表。这当然可以。但是我个人更偏向于，如果要用插件来解决房间中的某个问题，那么就在该房间中对任何问题都使用插件。当用户加入到房间时，他们会从插件那里获取一份初始用户列表，然后根据需要添加或删除事件。

下面这几个理由可以用来说明为什么我们要避免使用标准的用户列表行为。

- 在游戏中，当你最初了解一个玩家时，你可能会需要知道一些自定义信息，比如说用何种颜色来表示他以及他在游戏中的级别是多少。如果依靠房间中固有的用户列表，那么我们将得不到与玩家相关的那些自定义信息。
- 用同一块代码来解决所有重要问题，这样做会很整洁。我喜欢在一块区域内处理所有会发生的各种各样的行为。
- 根据你所创建的游戏的不同，可能会有多种类型的玩家。与更为刻板的内建用户列表所不同的是，插件允许一些玩家只是在游戏中旁观而不会被添加到活跃玩家的列表中。

6.4.5 DigGame类

此类包含该游戏中多数功能：创建新房间、与插件通信以及确定在客户端中所保留的少量游戏逻辑代码的数量。我们将关注一些过去未曾介绍过的用于与服务器端交互的代码。

1. ElectroServer 监听器

在 DigGame 类中只需两种监听器：

```
_es.addEventListener(MessageType.JoinRoomEvent,
➔ "onJoinRoomEvent", this);
```

```
_es.addEventListener(MessageType.PluginMessageEvent,
➔ "onPluginMessageEvent", this);
```

在第 5 章中我们介绍过 JoinRoomEvent 事件的事件监听器程序。在这里出现了一个新的 PluginMessageEvent 事件。当一条新消息从插件传到客户端时就将触发此事件。我们稍后来看看关于这两种事件的事件处理器的内容。

2. 用插件创建房间

创建房间的程序首先出现在第 5 章。正如我们提到过的那样，你需要在创建房间时设置很多选项。对本游戏以及本书中许多范例而言，创建房间时需要指定一个插件。

一旦创建了 DigGame 类，就会调用 joinRoom 函数来创建一个房间或使一个用户加入到一个房间中。这个函数包括了如下代码：

```
//create the request
var crr:CreateRoomRequest = new CreateRoomRequest();
crr.setRoomName("Dig Game Room");
crr.setZoneName("Dig Game Zone");

//create the plugin
var pl:Plugin = new Plugin();
pl.setExtensionName("GameBook");
pl.setPluginHandle("DiggingPlugin");
pl.setPluginName("DiggingPlugin");

//add to the list of plugins to create
crr.setPlugins([pl]);

//send it
_es.send(crr);
```

上述代码中被高亮显示的 CreateRoomRequest 配置部分是新知识点。首先，创建一个新的插件对象，接着我们给这个插件对象设置一些变量以确定我们要创建何种插件。这需要你指定扩展名称与插件句柄。插件名称是你要为房间中的插件新对象所取的名称。以后在传递消息给插件的时候会用到该名称。我经常会让插件句柄与插件名称保持相同，因为我根本不需要在同一房间中创建两个同样的插件。假如因为某些原因你想在同一房间中创立同一插件的多个对象，那么你就得保证每个插件对象的名称都是唯一的。

在创建并配置好插件对象后，我们会通过 setPlugins 方法将它添加到 CreateRoom-Request 对象中。setPlugins 方法包含一个插件数组。本例中我们只创建了一个插件，自然该数组中只包含一个元素。

CreateRoomRequest 对象被全部设置好之后，它就会被送往服务器端。房间创建好后（或者你已加入到已存在的房间中）就将触发 onJoinRoomEvent 函数。从这一刻起，你就可以和插件通信了。

- onJoinRoomEvent 函数

当你成功加入到房间后，onJoinRoomEvent 函数就被调用。以下是该函数的内容：

```
//store a reference to your room
_room = e.room;

//tell the plugin that you're ready
var esob:EsObject = new EsObject();
esob.setString(PluginConstants.ACTION, PluginConstants.
➔INIT_ME);

//send to the plugin
sendToPlugin(esob);
```

我们会存储一个对刚加入房间的引用并将其作为类属性来使用。

接下来我们将和插件进行通信——这在本书中可是头一次！我们告诉插件我们已经在房间中并准备玩游戏了。为此我们会创建一个新的 EsObject 对象，并且在其中设置一个 INIT_ME 行为。该行为不需要额外的属性。在创建 EsObject 对象并以必要的属性对其格式化后，我们将调用 sendToPlugin 函数，它将接受该 EsObject 对象作为其参数。我们接下来看看这个函数。

- sendToPlugin 函数

每当客户端需要往插件发送一些信息时，它都先创建一个 EsObject 对象并将其格式化，然后把它设置在一个要发送给服务器端的 PluginRequest 对象中。对于本书中绝大多数范例和游戏而言，客户端只需要和一个插件进行对话。既然这样，那么我们可以把大部分的用来发送给插件的代码都封装到单独的一个函数中：

```
private function sendToPlugin(esob:EsObject):void {
  //build the request
  var pr:PluginRequest = new PluginRequest();
  pr.setEsObject(esob);
  pr.setRoomId(_room.getRoomId());
  pr.setZoneId(_room.getZoneId());
  pr.setPluginName("DiggingPlugin");

  //send it
  _es.send(pr);
}
```

在上述函数中我们创建了 PluginRequest 类的一个新对象。接着我们在该对象中置入被调进该函数中的 EsObject 对象（包含自定义数据）。然后我们得指定这个请求消息发往的插件以及该插件所属的房间。一旦 PluginRequest 对象被全部配置好，我们就把它发送出去。

- DIG_HERE

现在让我们来看看怎么样告诉服务器端我们所要挖掘的位置。一旦检测到鼠标点击事件，

mouseDown 事件处理器就被调用，它包含如下代码：

```
if (!_trowel.digging && _room != null) {
  //tell the plugin you want to dig here
  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔ DIG_HERE);
  esob.setInteger(PluginConstants.X, mouseX);
  esob.setInteger(PluginConstants.Y, mouseY);

  //send
  sendToPlugin(esob);

  //animate
  _trowel.dig();
  playSound(new DIG_SOUND());
}
```

如你所见，主要的代码都封装在一个条件语句中。在告诉服务器端我们要在某个位置进行挖掘之前，我们要检查一下以确信没有正在播放挖掘动画且我们已经加入到一个房间中。_trowel 属性引用的是鼠标指针图标。如果正在播放挖掘动画，那么 _trowel.digging 返回值为 true。

如果你没有在挖掘并且在房间中，那么一个 EsObject 对象就会被创建并被格式化，接着会被发送给插件。这个对象中包含着一个 DIG_HERE 行为，它还指定了客户端要挖掘位置的 x 和 y 坐标值。

当该 EsObject 对象被发送给服务器端后，客户端会使铲子图标播放挖掘动画并随后播放一小段挖掘声效。

- onPluginMessageEvent 函数

onPluginMessageEvent 函数何时会被调用？你猜对了，是当客户端接收到从插件发来的消息的时候。我们在这里处理它的方式与在本书后续章节中处理方式相同：我们使用一个包含 action 变量的 switch 语句，这些 action 变量存在于 EsObject 对象中。然后对于每个 case 分支语句，我们都会把 EsObject 对象调入到在其相应 action 条件下所使用的自定义处理函数中。

下面是这个函数的代码：

```
public function onPluginMessageEvent(e:PluginMessageEvent):
➔ void {
  var esob:EsObject = e.getEsObject();

  //get the action which determines what we do next
  var action:String = esob.getString(PluginConstants.ACTION);
  switch (action) {
       case PluginConstants.DONE_DIGGING:
```

```
                handleDoneDigging(esob);
                break;
            case PluginConstants.PLAYER_LIST:
                handlePlayerList(esob);
                break;
            case PluginConstants.ADD_PLAYER:
                handleAddPlayer(esob);
                break;
            case PluginConstants.REMOVE_PLAYER:
                handleRemovePlayer(esob);
                break;
            case PluginConstants.ERROR:
                handleError(esob);
                break;
            default:
                trace("Action not handled: " + action);
        }
    }
```

PluginMessageEvent 事件对象包含着一个从插件传递过来的 EsObject 对象，以及发送该对象的房间的 ID 与该房间所属区的 ID，另外还有发送该对象的插件名称。在一个你有多个插件或者你加入到多个房间的程序中，你可能会使用房间 ID 和区 ID 以及插件名称去决定如何使用 EsObject 对象。

我们在上述函数中创建了一个叫做 action 的变量，并将 esob.getString(PluginConstants.ACTION) 赋值给它。这个变量在 switch 语句中的作用是将 EsObject 对象封送（marshal）给相应行为所对应的自定义处理器。DONE_DIGGING 行为是唯一针对本游戏的专有行为。其他的行为用来处理用户列表的更新以及可能会发生的错误。它们将会在以后多数范例中重复使用，其作用也与本例中你所看到的完全相同。

- handleDoneDigging 函数

当 onPluginMessageEvent 函数处理 DONE_DIGGING 行为时，handleDoneDigging 函数将被执行。每当有玩家完成一次挖掘，服务器端就会把 DONE_DIGGING 消息发送给房间中的所有玩家，并将玩家的新分值广播给房间里的所有人。如果你正巧是那个挖掘工并且找到了某样物品的话，那么我们会创建该物品并将它显示到你的屏幕上。

下面是从 EsObject 对象中取出玩家名称与得分值的代码（所有玩家都被管理在一个创建好的叫做 PlayerManager 的类中）。

```
//grab some initial information off of the EsObject
var name:String = esob.getString(PluginConstants.NAME);
var score:int = esob.getInteger(PluginConstants.SCORE);

//find the player and update the score property
var player:Player = _playerManager.playerByName(name);
player.score = score;
```

接下来，让我们看看如果玩家恰好是客户端正在管理的玩家时所运行的代码：

```
var found:Boolean = esob.getBoolean(PluginConstants.
➔ ITEM_FOUND);
if (found) {
  //get the id that says which of the 4 item types was found
  var itemId:int = esob.getInteger(PluginConstants.ITEM_ID);

  //create item, set its type, position it, and add to screen
  var item:Item = new Item();
  item.itemType = itemId;
  item.x = _trowel.x;
  item.y = _trowel.y;
  _itemsHolder.addChild(item);

  //play a positive sound since you found an item
  playSound(new FOUND_SOUND());
} else {
  //play a negative sound since you found nothing
  playSound(new NOTHING_SOUND());
}
```



注意　如果事件发生在某一个玩家身上，而这个玩家也可能是“你”，那么将会采取一些额外行为。本书中这种处理方式被反复用到。

我们从 EsObject 对象中抽离出一个布尔值用来判断是否找到物品。如果该布尔值返回值为 `true`，那么就意味着找到了一个物品并且在 EsObject 中也会有该物品的 ID 存在。由于游戏中有 4 种物品，所以存在 4 个可能的物品的 ID 值——0 ～ 3。接着会创建一个 `Item` 类的新实例并将 ID 值设置给它（如此，该实例就会知道选用那张图像来展示自己了），然后将其添加到屏幕上。

如果有物品被找到，就会播放一个表示肯定的音效；如果没有找到任何物品，就会播放一个表示否定的音效。

- 玩家列表行为

如本章前面所述，我们不选用房间内建的用户列表功能，因为我们将使用插件，我们可以用插件来处理任何事情。插件是万能的，因此它可以在客户端加入房间时为其提供完整的用户列表，并且当用户列表改变时为客户端提供更新。

为了避免在作为 ElectroServer API 中一部分的 `User` 类与内建的用户列表功能两者间发生混淆，我们将用 `Player` 类来管理连接的客户端，而且我们会把玩家的列表称作玩家列表。玩家列表中的每一项都是一个 `Player` 类的实例。玩家在一个被称作 `PlayerManager` 的类中被管理。`PlayerManager` 类能使你获得完整的玩家列表并能从该列表中添加或删除玩家，还可以让你按照名称查找玩家。

玩家初次加入房间时，客户端会向插件发送 INIT_ME 消息。插件会以 PLAYER_LIST 行为作为响应，该行为提供了房间中完整的玩家列表。从这时起，玩家会接受 ADD_PLAYER 与 REMOVE_PLAYER 消息以保持玩家列表为最新状态。

让我们看一下 handlePlayerList 函数的内容吧：

```
var players:Array = esob.getEsObjectArray(PluginConstants.
➥ PLAYER_LIST);
for (var i:int = 0; i < players.length;++i) {
  var player_esob:EsObject = players[i];

  var p:Player = new Player();
  p.name = player_esob.getString(PluginConstants.NAME);
  p.score = player_esob.getInteger(PluginConstants.SCORE);
  p.isMe = p.name == _myUsername;

  _playerManager.addPlayer(p);
}
refreshPlayerList();
```

第一行代码从调入该函数中的 EsObject 对象那里获取了一个 EsObject 对象数组。该数组中的每一个格式化 EsObject 对象都包含了房间内每一个玩家的信息——玩家姓名及其当前得分（见图 6-6）。因此，我们可以通过遍历该数组并处理每一个 EsObject 对象来创建玩家列表。当该过程结束时会调用 refreshPlayerList 函数。此函数获取玩家列表并将它显示到屏幕上的 List 组件中。

player132, score: 9700
player492, score: 36600
player864, score: 95300

图 6-6

6.4.6 服务器端代码

“挖宝”游戏的服务器端插件维持着全部的游戏状态。这些状态包括每一玩家的信息以及已经被挖掘过的位置信息。这里会用到以下 5 种用 Java 编写的类。

- DiggingPlugin 处理游戏逻辑以及与客户端的通信。
- PluginConstants 与客户端的 PluginConstants 类的使用方式相同。
- PlayerInfo 是一个用来存储游戏中任何单个玩家信息的对象类。
- ItemType 是一个 Java 枚举类，用来指定挖掘中所能找到物品的不同种类。
- Grid 是一个数据结构，它被用来记录每一个已被挖过的位置。

现在让我们来看一看 DiggingPlugin 类中的 7 个主要方法。



注意　在目录 book_files/examples_extension/server/src/com/gamebook/digging 下可以找到完整的源代码。

1. Init

该事件处理器在每次创建 DiggingPlugin 对象时都会被调用。我们最好把那些需要在客户端进入房间之前被执行的代码（比方像初始化数据结构和变量这样功能的代码）添加到其中。

2. userEnter

该事件处理器在用户每次尝试进入房间时都会被调用。更复杂的游戏会在此处设置游戏逻辑代码以决定是否允许用户进入房间。对于 DiggingPlugin 类而言，我们只是简单地记录下用户已经进入房间这个事实，并假设用户会发送一个使其稍后被初始化的请求。

3. userExit

每当有用户离开房间时都会调用该事件处理器。对于 DiggingPlugin 类而言，我们会从记录玩家信息的数据结构（playerInfoMap）中删除用户，然后广播一条消息给剩余的玩家。在其他游戏中，当剩余玩家太少时，我们需要决定游戏是否应该在此时结束。回合制游戏尤其需要在 userExit 事件处理器中设置些清理代码以确保游戏在某个玩家离开后不会中止。



注意　这里所列出的前五个方法实际上是用于捕捉房间或基于服务器端的事件的事件处理器。

4. Destroy

当插件被销毁时也即最后一个用户离开房间后，Destory 事件处理器就会被调用。这个方法可以取消任何预定回调（scheduled callback）。DiggingPlugin 类使用预定回调来处理在客户端发出“dig here”（挖掘这里）的请求与服务器端发送挖掘结果之间存在的 2 秒钟等待时间。

5. Request

此事件处理器是大多数插件的核心部分。它是客户端 PluginMessageEvent 事件的服务器端模拟。它处理客户端发过来的每一个插件请求。在“挖宝”游戏中只存在以下 3 种用户能请求的行为。

- INIT_ME 用来告知服务器端，客户端已经接收并处理了 JoinRoomEvent 事件。服务器端会调用 handlePlayerInitRequest 方法来通报有新玩家加入，并把新玩家添加到 playerInfoMap 数据结构中，然后将房间中所有玩家的列表传给这个新玩家。
- POSITION_UPDATE 会随消息转述给所有玩家，同时一齐被转述的还有添加进消息但未

被服务器端所额外处理的玩家姓名。

- DIG_HERE 用来告知服务器端，客户端想要在指定位置挖掘。服务器端调用 `handleDigHereRequest` 方法来检查该位置是否允许挖掘以及是否有玩家正在挖掘中。如果该位置允许被挖掘，那么会使用预定回调来设置一个计数器，它随后会在 2 秒钟等待之后向所有玩家发送一条消息。

6. sendAndLog

我们用 sendAndLog 方法发送一条消息给所有在房间中的已初始化用户，并且记录下这条发送的消息。例如，每当用户结束挖掘时，客户端就会接收到该方法发送的关于挖掘结束的消息。

7. sendErrorMessage

该方法会为指定的玩家发送（并记录下）一条包含错误情况的消息。它有助于使客户端知道可能会导致的错误，并且通过记录下错误事件方便后续分析。

第 7 章

实 时 运 动

可以使用 Flash 创建很多类型的多人游戏，比如纸牌游戏、休闲游戏，还有那些需要玩家操控角色实时运动的游戏（比如赛车游戏、射击游戏或者虚拟世界），等等，不胜枚举。我曾经在同事中做过一次非正式调查，问他们假如有机会的话他们会尝试制作哪种类型的多人游戏。他们大多都说想做射击游戏、格斗游戏、赛车游戏、RPG 游戏（角色扮演游戏），还有人甚至想做即时战略游戏。而如果你想做好这些类型的游戏，并提供有趣的用户体验，那么一个必不可少的要素是能让一个或多个游戏对象实时运动。

我在 Google 上搜索了近 90 分钟，发现只有少数几个 Flash 多人游戏能满足以上这些要求。你可能会觉得纳闷：既然实时游戏如此吸引人，为什么相关的游戏却如此少见呢？我觉得有两点原因。首先，也许你很想开发这类游戏，但客户却没有这方面的需求（换句话说，没有人为你投资）；其次，缺少相关的学习资源。有些关键技术很难找到相关的参考资料，比如说实时运动（real-time movement）、网络延时隐藏（latency hiding）以及预测运动（predictive movement）。因此游戏开发者们被迫要靠自己来解决这些问题。

本章讲述的概念是本书最重要的一些内容。掌握了这些知识，当你需要在多个客户端之间实现同步运动的时候，就能确保运行效果平滑流畅。而且即将介绍的代码与概念也将在本书后续几章（尤其是第 9 章和第 10 章）中被多次用到。

本章中，我们将会了解引导游戏对象运动的几种路径类型，以及路径信息是如何在服务器端和客户端之间来回传递的。另外还将介绍几个概念——基于帧的运动、基于时间的运动以及网络延时。最后，我们将借助基于时间的运动，了解一下在隐藏延时中使用的预测运动的几种方式，然后用两个范例来演示其中的一种方式。

7.1 响应控制

大部分多人游戏都属于服务器端权威型。如果没有得到服务器的允许并对运动进行验证的话，客户端是不能擅自移动它的角色、汽车或船舶等东西的。在开发第一个实时角色运动游戏（一个侧视角平台游戏）时我曾犯过一个错误，即总是等收到服务器响应之后才开始运动角色。我捕获键盘的输入并将其发送给服务器，真正把客户端当成是虚拟终端了。服务器分析键盘的

输入值，然后将计算得到的新运动信息返还给客户端。这虽然能够实现同步，但玩起来会让人感觉厌烦。由于客户端要等待服务器的响应，所以会产生一个键盘响应延时。如果游戏运行在本地网络中，延时倒还不是很明显，但如果游戏是在互联网上运行的话，这确实是个问题。

假如游戏是基于时间同步的，而且客户端认为应该可以避免服务器端对运动与碰撞进行验证，那么最好的用户体验就是让客户端立即响应用户的输入，就好像服务器会同意这样做一样。如果后来服务器同意了（几乎都会同意），那么客户端和服务器端也就达成了同步。假如万一服务器拒绝了运动请求，那么客户端就应该对之前的操作予以修正，在此过程中会导致跳动的现象发生。这种罕见的跳动是可以接受的，因为从总体上来看，响应还算令人满意，用户体验也还不错。

7.2　路径类型

想想下面这些常见的沿着路径实时运动的游戏对象。

- 沿着直线运动的导弹。
- 虚拟世界中围绕着游戏物件走来走去的化身。
- 第一人称射击游戏中的角色。
- 赛车游戏中的汽车。

尽管上述游戏对象都要以实时的方式在屏幕上运动，但首先它们用于确定运动路径的技术就各有不同。下面让我们看看其中的几种方法。接着我们会讨论什么是视线，以及如何用视线来约束路径。

7.2.1　路点

路点（waypoint）就是游戏对象的目标位置。游戏对象会从其当前位置运动到该路点。当然也可以提前创建多个路点。例如在一个即时战略战争游戏中，你可以提前设立多个路点，用它们来引导你的车辆沿指定路线行进。车辆从其当前位置出发，依次经过每一路点就会到达最终路点。

在区块式虚拟世界中，如果使用 A* 这样的寻路算法，它就会返回一个区块列表，其中的区块构成了最终生成路径。而路径中的每一个区块你都可以看作是一个路点（见图 7-1）。

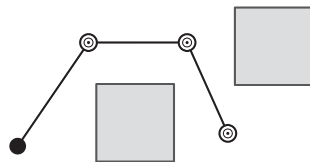


图 7-1　路点

7.2.2　矢量/航向

我们大概都玩过第一人称射击游戏（FPS），通常在 FPS 游戏中，你能控制角色的跑动、行走与跳跃。你可以使角色起步、停下，并且以非常快的速度改变方向，我们将这种类型的响应控制称作“抽筋式”。

想想看，在 FPS 游戏或赛车游戏中，某个玩家可能会飞快地操控角色或赛车，以至于其他玩家很难正确并顺利地推断出它们的位置以及它们随后的去向。要想将你的方位通知给其他玩家，首先要考虑的就是发送你的位置信息。尽管对于许多慢节奏的游戏来说，基于这样的一些做法确实能够起作用（我们会在后面的范例中予以演示），但对于快节奏的游戏来说，玩家却不可能收到足够多的信息来平滑且合理精确地渲染出所有的游戏对象。

矢量（vender）能帮助我们解决这个问题。矢量是数学中的概念，它由一个数值和一个方向组成。比方说，30 英里 / 时[①]的东风就构成了一个矢量。你可以用这个概念来考虑一下赛车游戏中的赛车，在任何给定时刻，赛车都是按照某一矢量（速率与方向）在运动。在一个快节奏的抽筋式游戏中，对于所有运动中的对象而言，知道它们在某时的矢量要比只知道它们此时所在位置有用得多。

另外，我发现有时将游戏对象的位置同矢量结合起来会很有用。在一个*航向*（heading）中，我会组合使用矢量信息、位置信息以及一些其他信息（本章后面将会讨论），如图 7-2 所示。航向是对运动中的游戏对象状态描述得最完整最及时的信息。

知道了游戏中所有运动对象的航向信息，你就可以执行*预测运动*（predictive movement）了。也就是说，在接收到更多最新的服务器响应之前，它能让你的游戏对象在最合理的方向上继续运动。

7.2.3　视线

视线的概念非常简单。假想有一个初始点，比如说一辆车或者一把枪所在的位置。视线就是从初始点指向某个选定的向外的方向（比方说枪口所面对的方向）所引出的一条路径，并且最终当它碰到某个物体时就会停下来（见图 7-3）。

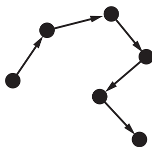


图 7-2　航向

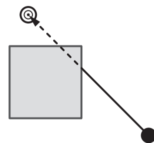


图 7-3　视线

游戏中的人工智能（AI）使用视线来确定它（AI）所能看到的场景内容，就好像它也是游戏玩家中的一员一样。例如，视线可以被用来在游戏中将枪对准某物体，或者它也可以用在车辆或角色随鼠标单击而运动的游戏中，目的是要使游戏对象沿直线路径行进。在像这样的游戏中，视线有助于我们确定期望路径或尝试路径是否有效。或者，在玩家建立一系列的路点时，视线也能帮助确定路径的每段是否有效（如果路径中的任何一段阻碍了游戏对象的运动，那么该路径就是无效的）。你会在第 9 章中用到这些概念。

① 1 英里 =1.609 344 km，1 英里 / 时 =1.609 344 km/h。——编者注

7.3 基于帧的运动

对于运动中的游戏对象而言，有两种方式可以更新它们的位置：基于帧间隔或基于时间间隔。当使用基于帧间隔的方式时，我们将完全不考虑时间变化，对象的位置会随着帧的刷新而不断变化。例如，每经过一帧，飞行导弹的位置就会在 x 轴正方向上增加 3 像素，在 y 轴正方向上增加 6 像素。

基于时间运动则是根据经过的时间来更新游戏对象位置的。即便是这样，每经过一帧，我们也需要在屏幕上直观地更新游戏对象的位置，只不过那些更新的位置是由经过的时间来确定的，而与经过的帧数没有关系。我们将在本章后面内容中详细地讨论基于时间的运动。

7.3.1 何时使用基于帧运动

或许对于你可能开发的任何一款单机游戏来说，使用基于帧运动的方式会很不错。但通常在多人游戏中，基于帧运动并不是最好的选择。不过在像类似桌球这样的回合制游戏与多数社会化虚拟世界中，基于帧运动也能很好的表现运动中的游戏对象。

在上面的两个例子中，运动中的游戏对象在到达某个特定位置的准确时间并不是那么重要，重要的只是该对象经过了所有预定的路径并最终到达了正确目的地。由于每个客户端的帧速率都有所不同，这就造成了播放相同的动画，在某个客户端上需要的时间可能比在另一个客户端上要长得多。不过，最终所有的客户端都能完整地将动画播放完。

让我们来看一下使用基于帧运动而不够理想的情况。设想有一个使用基于帧运动方式的双人射击游戏。子弹以水平方向被射出，每一帧都会移动 8 个像素。由于硬件的不同，并且由于这种游戏一般都很紧张刺激，所以会造成游戏在双方机器上运行有着显著的差异，玩家 1 的机器能达到每秒 30 帧，而玩家 2 只能达到每秒 25 帧。每经过 1 秒钟，子弹在玩家 1 的机器上就会移动 240 像素，而在玩家 2 的机器上只能移动 200 像素。游戏运行时间越长，玩家间就越不同步。如果当子弹击中玩家时服务器端会向客户端发送事件以表明该情况，那么这种不同步的情况就会缓解一些，但对于电脑较慢的玩家（在这里指的是玩家 2）来说，这种游戏体验依然不够理想。

总的看来，基于帧运动通常适用于对时间同步要求不太高的多人游戏。

7.3.2 当前位置：Here I am

本节中，我们来看一个基于帧运动的多人游戏对象移动的范例，在其中我们只考虑游戏对象的位置。我把该方法叫做“Here I am”（我在这里）——每个游戏对象都在不停地说“我在这里，我在这里，我在这里”。游戏对象并没有说将要往哪里去，只是说它在什么地方。



注意 在 book_files/chapter7/hereiam 目录中可以找到“Here I am”的范例文件。

这个范例的 Flash Develop 项目文件名为 HereIAm.as3proj。为了测试这个范例，你需要启动 ElectroServer，加载并编译这个项目。跟我们所有范例一样，它也会加载一个有连接设定的 XML 文件。

“Here I am”方法的优缺点

这种方法的好处是易于编程。在时间同步不很重要的情况下它非常好用。一个简单范例是显示所有其他在线用户的鼠标位置。如果你想与应用程序进行交互并想看到其他所有在线用户的光标在屏幕上运动，那么采用这种方法就很理想，因为在这种情况下光标运动方式与位置的精确度并不是太重要。

而这种方法也有个主要的缺点，那就是：在某一时刻，运动中的游戏对象永远不会出现在它此刻应该在的位置。游戏对象的运动总是会有些延时，这是因为同步消息到达客户端之前需要一些时间。



在这个范例中，你要使用方向键来控制一个行走的外星人。你还能看到碰巧也在这个房间中的其他玩家的外星人的行走情况（你可以通过打开多个 SWF 副本文件来进行测试）。当你的外星人运动时，你会发现有个渐隐的复制影像跟在它后面，我们称这个影像为镜像（mirror）。你在本地机器上控制你的外星人，而镜像则是由服务器端所返回的信息产生的。镜像的作用就是让你看到你的外星人在其他玩家屏幕上的位置。

由于镜像所做的是匀速运动，所以它在运动过程中永远也跟不上你的外星人。而且事实上，当出现少量延时或者帧速率下降时，镜像与外星人的距离还会越来越远。不过当外星人停止行走后，镜像终会赶上它并停下来。

该范例只演示了多人游戏中的一些基本行为：检测何时添加或删除用户，以及将 EsObject 对象播发给同一房间中的每个玩家。因为这些都很基础，所以不需要用到插件，我们只用房间对象的默认事件就能实现所有功能。我们不使用插件，而是把 EsObject 对象附加到一个 `PublicMessageRequest` 事件中。如果你想根据这个范例来开发游戏，你就需要加入插件以使游戏状态能被服务器端管理并验证。无论是使用插件还是默认的房间事件，EsObject 对象的格式化以及客户端处理位置更新的逻辑都是一样的。

当该程序启动的时候就会建立一个连接，随即用户以一个随机名称登录并随后加入一个房间中。我们将会为 `JoinRoomEvent`、`PublicMessageEvent` 和 `UserListUpdateEvent` 事件创建事件监听器。

我们来看一下如何添加你的外星人、如何发送外星人的位置更新消息以及怎样处理接收到的位置更新消息。

这里是 `onJoinRoomEvent` 的事件处理器：

```
public function onJoinRoomEvent(e:JoinRoomEvent):void {
  _room = e.room;

  var guy:Guy = new Guy();
  guy.x = 200;
  guy.y = 200;
  guy.playerName = _es.getUserManager().getMe().getUserName();
  _myGuy = guy;
  addGuy(guy);
}
```

首先，我们把对刚加入房间的引用储存到一个类属性。然后创建一个 Guy 类的新实例用来代表你所控制的外星人。Guy 类包含了外星人的图像与位置信息。这个实例刚开始时被放置在坐标（200,200）处，然后我们再把你的名称提供给它，接着将其存储为类属性 _myGuy，最后将它传递给 addGuy 函数。addGuy 函数会存储所有对 Guy 的引用以便将来查找。

这个范例中有一个逐帧触发的事件，它用来处理运动逻辑、键盘捕获以及发送更新消息：

```
private function enterFrame(e:Event):void {
  if (_myGuy != null) {
       checkKeys();
       moveGuys();

       //send a position update every 500ms
       if (getTimer() - _lastTimeSent > 500) {
            sendUpdate();
       }
  }
}
```

首先我们用一个条件语句来判断你的外星人朋友是否存在。如果存在，我们就调用 checkKeys 函数，此函数能检测方向键的状态并在 8 个方向中的一个方向上运动外星人（如果你没有按下任何键，那就没有任何运动）。moveGuys 函数遍历屏幕上所有的外星人，让他们从当前的位置往目标位置运动一步。然后，每过 500 ms，你的外星人的位置更新信息就要发送给其他所有人。当然发送更新信息的频率可以更智能一些，比如当经过 500 ms 后位置确实发生了变化再发送。

现在我们来看看这个 sendUpdate 函数：

```
private function sendUpdate():void {
  //format the EsObject to send
  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔UPDATE_POSITION);
  esob.setInteger(PluginConstants.X, _myGuy.x);
  esob.setInteger(PluginConstants.Y, _myGuy.y);
  esob.setString(PluginConstants.NAME, _myGuy.playerName);

  //send the EsObject via the PublicMessageRequest
```

```
    var pmr:PublicMessageRequest = new PublicMessageRequest();
    pmr.setRoomId(_room.getRoomId());
    pmr.setZoneId(_room.getZoneId());
    pmr.setMessage("");
    pmr.setEsObject(esob);

    _es.send(pmr);
}
```

该函数会将你的外星人的位置更新信息发送给房间中的所有人。我们首先创建一个EsObject对象来存储自定义信息，然后在其中设置一个`UPDATE_POSITION`行为变量，这也是本例中唯一用到的行为类型。接着把外星人的x与y坐标值及其名称一同放入EsObject对象中。如第5章所讲的那样，本例中我们要用`publicMessageRequest`请求对象将EsObject对象发送给房间里的所有用户，而这就需要我们创建一个`PublicMessageRequest`类实例并且填充进必要信息。

当客户端收到该消息后，就会触发`onPublicMessageEvent`函数，该函数会立即截获EsObject对象并把它交给`handleUpdatePostion`函数处理。此函数会从EsObject对象中获取玩家姓名以及外星人的x与y坐标值。然后根据玩家姓名定位到与他相应的`Guy`实例（如果不存在就创建一个新的），接着用下面的代码来更新目标位置：

```
if (!guy.isMe) {
  guy.walkTo(x, y);
}
```

你只能调用除了自己之外的`guy`对象的`walkTo`函数。该函数为`guy`对象设置一个新的目标位置，然后外星人就会在每一帧中不断地去靠近那个位置，直到它抵达该位置或者它又接收到了新目标位置。

你会发现自己的外星人角色会根据方向键的状态立即更新位置，而无需等待服务器端的响应。以后你会看到更多这样的范例。假如你为应用扩展了一个服务器端插件，它就会对你发送的运动与位置更新信息进行验证，以确保它们能被客户端正确地处理。

服务器端更新：间隔多长时间

在这个外星人范例中，我们为什么要选择每500 ms发送一次更新信息？为什么不是5 ms或者5 000 ms呢？这是由于，对于那些在实时多人Flash游戏中经常更新状态的对象而言，我们并不希望向服务器端过快地发送更新信息。多快才比较合适呢？依我的经验来看，每秒2～3次发送更新信息就比较合适——如果超过这个频率，发送再多的信息也并不能让结果看起来更好，而且毫无裨益。在这个范例中，500 ms的频率足以使其他玩家看到你的外星人在受你的操控。

7.4 网络延时与时钟同步

本章的终极目标就是介绍如何在多个客户端上实现平滑的、基于时间同步的运动效果。而其中一个关键性的问题是，如何让所有客户端都认同当前的时间。这是一个超乎你想象的更具有挑战性的任务！本节我们将讨论有关 ping 与网络延时（latency）的内容，并深入研究下如何得到最接近真实环境下的网络延时，还将介绍如何使用 `Clock` 类计算出真实的服务器时间。

7.4.1 ping和网络延时

大多数人应该都了解 ping 是什么意思，它指的是一条简单消息从客户端发送到远程端点然后再返回到客户端所用的总时间，或者只用来指代那种向任何地点发送简单消息的行为。在本书中，我们用 ping 来表示一个简单消息在客户端与服务器端往返一次所用时间（图 7-4）。



注意 按照维基百科（Wikipedia）上的解释，我们这里给出的 ping 定义是不正确的。但就一般的游戏开发而言，这个定义还是能被接受的。

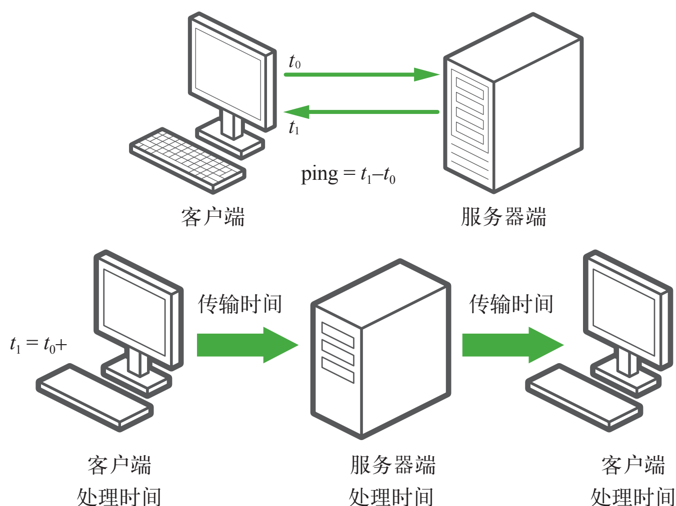


图 7-4　ping 是消息往返一次所需的时间，在这里就是 t_1-t_0

图 7-4 给出了消息在一次往返过程中会占用时间的主要处理过程。如果你想计算 ping 值，首先你要知道发送请求的时间点，还需要知道响应返回的时间点，然后将两值相减即可。ping 的大多数时间都花费在互联网的传输过程中，不过另外一些过程也会占用一些时间，这些过程主要包括：应用程序尝试发送消息、服务器端处理入站消息并且接着发送另外一条消息给客户端、客户端随后处理最终收到的入站消息。在客户端和服务器上处理消息所花费的总体时间一般都低于 1 ms，而消息在因特网上传输所需的时间一般都在 700 ms 到 200 ms 之间。正因为处

理时间与传输时间相比可以忽略不计，所以我们认为 ping 指的就是消息在互联网上传输的时间（见图 7-5）。

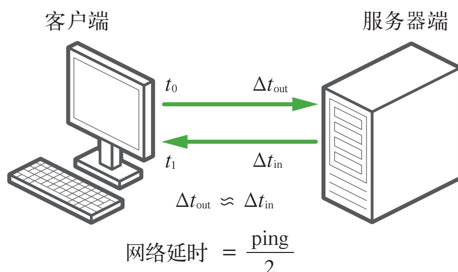


图 7-5

这里所使用的术语“网络延时”指的是消息从服务器到达客户端所需要的时间。当测算 ping 值时，我们无从得知消息是否从客户端到服务器端就花费了 90% 的传输时间，而从服务器端到客户端仅花费 10% 的传输时间，或者刚好都是 50%，再或者是其他比例。你所知道的只是消息发送时刻以及消息返回时刻。为了实现消息同步的效果，我们必须做出一个假设：ping 的时间一半花在前往服务器端的路上，一半花在返回客户端的路上。根据这个假设，网络延时的时间就是 ping 值的一半。

为了能在所有客户端上实现同步运动，客户端必须对当前时间达成共识。但是我们不可能按照每个客户端自己的时间来同步，而只能统一以服务器端的时间来同步。在本书我们所使用的技术中，客户端先发送一个 ping 请求，接着服务器返回一个含有毫秒级时间标签的响应，随后客户端用下面这个简单的公式就能测算出服务器端的时间：

```
server time = client time + offset
```

而上面的 `offset` 应该为

```
offset = server time - client time + latency
```

请注意，第二个公式的“server time”为 ping 返回的服务器时间标签，“client time”为响应到达时的客户端时刻，而“latency”为 ping 值的一半。

看了上面的公式，你就会发现正确的 latency 值是多么重要。server time 值是直接从 ping 响应中传递过来的，client-time 的值是从客户端本地系统时间中取得的。而 latency 值则需要尽可能准确地测算，当然这种测算过程有时也包含了一些假设成分。如果测算得出的 latency 值与其实际值相差太远，那么你会得到一个错误的服务器时间。这个差距越大，我们的游戏体验就会变得越糟。



注意 网络延时测算错误很可能是由一些很小的暂时性因素造成的，这会使得消息到达服务器端或者返回到客户端所用时间稍微长一些，从而网络延时就会被高估（因为我们假设的是传输来回过程都花费相同时间）。

那么我们怎么样才能提高网络延时测算的可信度呢？我所采用的方法是受一篇文章的启示，这篇文章名为 Minimizing Latency in Real-time Strategy Games，由 Jim Greer 与 Zachary Booth Simpson 所写，收录于 Charles River Media 公司出版的 *Game Programming Gems 3* 一书中。我们要多次测算网络延时值。在本书给出的代码中，默认情况下我们会测量 10 次。你第一感觉可能会想：只要计算这些数的平均值就行了。这当然不错，但仅仅如此还不够。当你进行多次测算时，有可能会出现因数据包丢失而自动重发的情况（这个不由你控制，而是由底层协议来处理的），或者有可能出现其他情况，导致数据包在服务器端的入站时间与出站时间相差较大。通过把每一个当前测得的数据值跟整个数据组的中间值进行比较，我们就能最大限度地检测出不正常的数据值（如图 7-6 所示）。

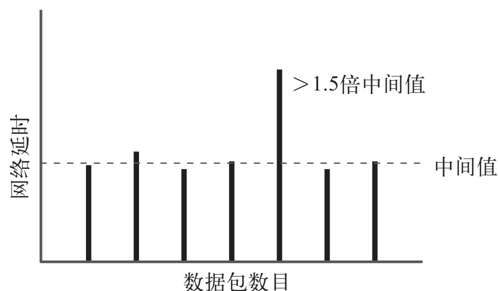


图　7-6

图 7-6 显示一个有 7 个测量值的数据组范例，你应该很容易看出有一个不正常的数据。如果某个测量值大于中间值的 1.5 倍，我们就可以断定该数据有问题然后舍弃它。剩下的当然就是有价值的数据了，此时我们就可以信心十足地说，它们的平均值就是最终的网络延时值。

7.4.2　使用Clock类

你可以在 com.gamebook.utils.network 包中找到 `Clock` 类，本书后面的内容也将经常用到这个类。为了使用 `Clock` 类，你只需简单创建一个它的实例，传入一个 ElectroServer API 的引用以及所使用的相关插件的名称，接着让它运行，如下所示。

```
_clock = new Clock(_es, "TimeStampPlugin");
_clock.start();
_clock.addEventListener(Clock.CLOCK_READY, onClockReady);
```

按照默认设置（上面所使用的范例），客户端将会向服务器发送 10 次请求，然后根据前面介绍的算法给出一个比较精确的网络延时值。当这一切完成后（一般不会超过 1 s），`Clock.CLOCK_READY` 事件就会被触发。然后你就可以像下面这样获取服务器的时间了。

```
_clock.time
```

以前你可能使用 `Date` 类或者 `getTimer` 函数获取时间，而现在你只需要全部替换成 `_clock.time` 就可以了。



注意　你可以在 book_files/chapter7/clocksync 目录中找到该技术的范例。

7.5　基于时间的运动

前面我们已经介绍了时钟同步，接下来让我们来看看基于时间的运动。如前所述，对于那些需要在多个客户端和服务器之间实现同步运动的游戏来说，基于时间的运动同样是一个非常重要的概念，这些游戏包括赛车游戏、射击游戏还有 RPG 游戏。

本节我们将了解基于时间进行运动（可预测的运动）的公式、网络延时隐藏的概念、`Converger` 和 `Heading` 类的使用，另外还将介绍两个基于时间运动的范例。

7.5.1　运动公式/可预测的运动

当游戏对象基于时间运动时，我们就会涉及物理学中的那些经典的运动学公式。给定对象当前的位置、速度以及加速度常量（可能为零），由这些公式就可以算出未来某个时刻对象的位置（图 7-7）。当然本书不是专门讲物理学的书，所以我们也不会这方面花很多时间。

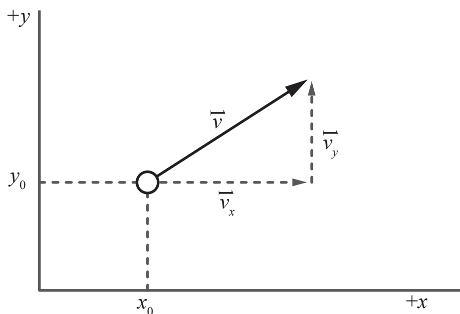


图　7-7

```
x = x0 + vx*t + (1/2)*ax*t^2
y = y0 + vy*t + (1/2)*ay*t^2
```

由以上基本运动学公式可知，在其他因素不变的情况下，如果知道对象当前的位置、速度及其加速度，那么就能够算出未来某个时刻对象的具体位置。一旦你接收到描述对象位置及其运动方向的信息后，你就可以继续让这个对象不断地运动下去。根据前面给定的信息让对象沿着某条路径运动的行为，就是我们前面多次提到的“可预测的运动”。



注意　物理学中的“加速”指的是任何时刻速度发生变化的这样一种状态。速度是矢量，它包含着角度和速率。故而当角度和速率发生变化时，我们就把它称为“加速”（甚至可以说减速也是“加速”的另一种表现形式）。

7.5.2　网络延时隐藏

基于时间的运动对游戏来说很有用。你可以一次处理运动中的很多游戏对象，并将其显示在所有收到最新信息的客户端的同一位置上。不过，在游戏中使用基于时间的运动也会带来一些挑战。我们在本节讨论其中一方面，7.5.3 节再来讨论另一方面。

假如使用运动学公式来确定游戏对象所应处的确切位置，那么也可能会在接收航向更新时产生一定的网络延时，而这就会导致运动中的游戏对象的连续位置发生中断。比方说你有两个客户端 A 和 B，A 在水平方向发射了一颗子弹，速度为 100 像素 / 秒。假如“发射子弹”的消息在到达 B 时用去了 150 ms 的时间，那么子弹在从被 A 发射出到 B 接收到信息的这段时间里共运动了 15 像素。所以当子弹被添加到 B 的屏幕上时，它应该按照运动学公式被放置在正确位置，也就是位于枪口右边 15 像素的地方（见图 7-8）。

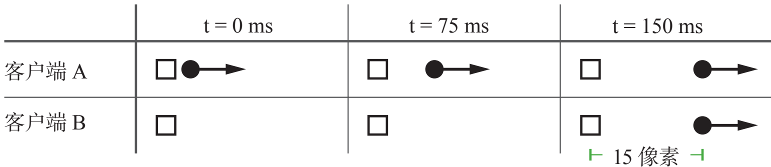


图 7-8

如同上面所描述的那样，两个客户端都需要精确的显示子弹的位置。这对于子弹来说或许不那么明显，不过要是你正驾驶着一辆赛车，或者你正在 FPS 游戏中来回地奔跑，情况就会大不相同了，游戏对象的加速度越大，它在客户端屏幕上产生的位置跳跃（也称作“抖动”）也就越大。不过我们得再次重申，汽车或者角色在屏幕上这样地抖动其实是我们所期望的。当收到最新消息时，客户端必须依据消息在传输过程中花费的时间把对象更新到最新状态。如果网络延时为 0，当然也就不可能有抖动的现象发生。

网络延时隐藏（latency hiding）技术就是用来解决这个问题的。这项技术通过尽量减小网络延时所带来的影响，为我们提供一种更为顺畅的游戏体验。本章中我们只介绍如何使用网络延迟隐藏技术来解决与运动有关的问题（在工业上也被称为“航位推测法”）。我们的目标是：在预测运动中加以平滑地修正航向，以使对象航向逐步收敛到正确航向。

要解决这个问题就需要找到这样一种方法，它能把对象当前航向平滑地过渡到刚接收到的最新航向。如果收敛的过程太慢，屏幕将因不能以足够快的速度更新而达不到我们所认为的同步效果，不过运动的过程将是非常平滑的；如果收敛的过程太快，屏幕上的对象会更接近它们应该在的位置，但同时也会在更新时导致对象剧烈地抖动。根据游戏而定，你得保证在没有出现剧烈抖动的情况下，调整平滑算法以使对象的航向尽快地收敛到正确航向（见图 7-9）。

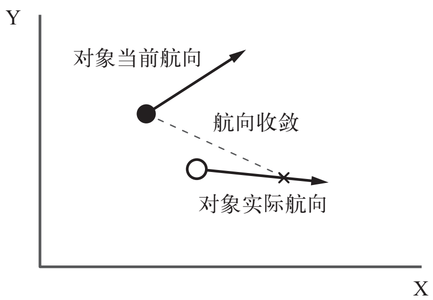


图 7-9

图 7-9 中的实心点代表客户端当前所看到的对象航向。空心点代表客户端刚刚收到的对象新（实际）航向。虚线代表的是从之前的航向平滑过渡到新航向的预测截取路径。

让我们来看看 4 种实现平滑过渡航向的方法。

4 种实现航向平滑过渡的路径方案

对于以下 4 种路径方案，我们都假设在不远的未来某时（通常是在几百毫秒以内），对象都会从它当前的航向过渡到最新位置的航向。每幅示意图中都有一条垂直虚线，它代表的是完全收敛到正确航向时的时刻。

从图 7-10 中你会看到，对象为了追赶正确的航向而以恒定加速度改变速率，直到与目标位置匹配为止。这似乎并不是一个好的解决方案，因为航向收敛前对象速率要远远大于收敛后的速率。这会导致对象突然加速，而后又突然减速。

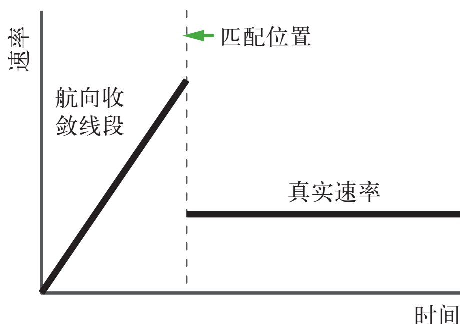


图 7-10　方法 1

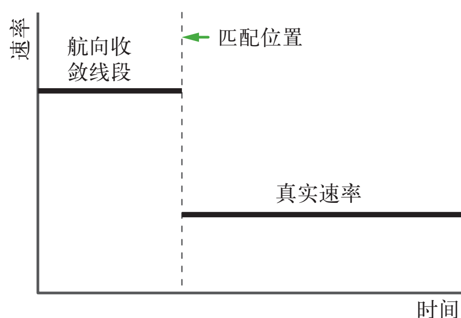


图 7-11　方法 2

第二种方案显示了对象以固定的速度实现航向收敛的过程（见图 7-11）。这看起来有点类似于第一个方案，也会导致剧烈的速率变化。尽管如此，假如游戏对象并不会有剧烈加速的话，这种方案也能良好运作，即运动的过程没有明显的抖动，画面比较平滑。因为这种方案简单实用，所以我们在后面的范例中就用这种方案。

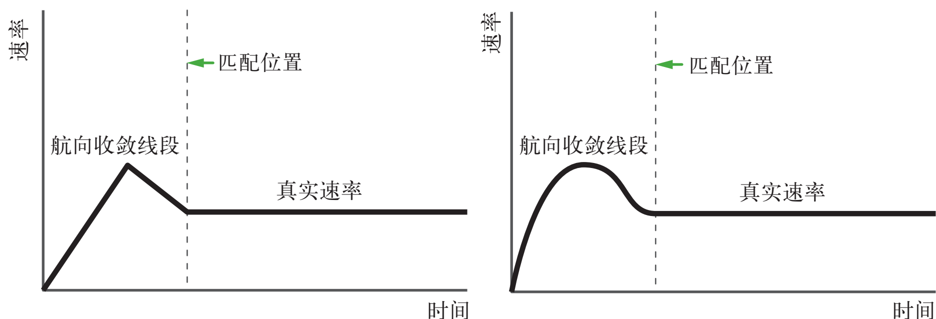


图 7-12　方法 3 与方法 4

最后的两个方案非常相似（图 7-12）。它们都尝试解决前面两个方案中速率剧烈变化的问题。通过平滑地改变速率，不断地调整当前的航向从而收敛到正确的航向。当收敛过程完成时，速率刚好和目标速率一致。方案 3 可以用运动学公式来实现，但方案 4 很可能得用贝塞尔曲线数学理论来实现。对于实现网络延时隐藏而言，应用贝塞尔曲线应该是最好的方案，它可以让画面看起来更加真实和更加平滑，当然它所需要的计算量也是最大的。

7.5.3　加速度

当使用基于时间的同步运动时，你还得考虑另一种情况。设想在游戏中玩家 A 与玩家 B 各自控制着自己的赛车，玩家 A 以每秒 100 像素的速率向右行驶，玩家 B 的屏幕上可以完美地显示玩家 A 的赛车的运动。接着玩家 A 紧急刹车，没有任何的打滑过程就彻底在原地停住。假如这个消息传递到玩家 B 花了 150 ms 的时间。当玩家 B 收到这个消息时，在他的屏幕上所显示的玩家 A 的赛车已经（由于预测运动）向前额外地运动了 15 像素。

你可能会想：“正好，我们刚才谈到的那些平滑处理技术能派上用场了。”不过你只说对了一部分。平滑处理能保证赛车停在正确的位置，但必须让赛车倒回一段距离。

图 7-13 描述了这个问题。实心圆代表玩家 A 的赛车在客户端 B 上的实时运动情况。而空心圆呈现的则是玩家 A 的赛车的真实航向。

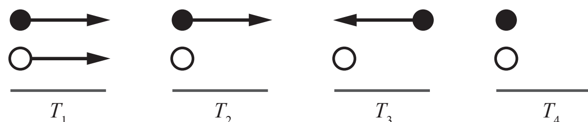


图　7-13

解决这个问题的方法是利用加速（请记住，减速也是一种“加速”）。如果玩家 A 决定停下他的赛车，那么一定要让其至少花费 500 ms 的时间慢慢停下来，而不要立即停下来，这样做可以让其他玩家有足够的时间接收到它的停止消息，并且使赛车在正确的时间点停下来（如图 7-14 所示）。

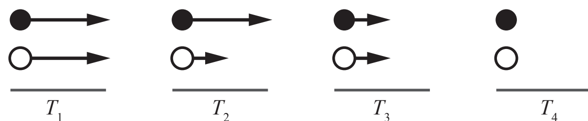


图　7-14

这里你所看到的就是如何利用加速度来实现的。由于网络延时，每个客户端的加速度都会作相应的调整，以保证它们的最终速度（这个范例中的速度为 0）都会在正确的时刻变得一致。接下来的两个范例就都用到了加速度。

7.5.4 Heading类 和 Converger类

这两个类（都是由我开发的）用来处理平滑运动会很不错。它们都存放在 com.gamebook.utils.network.movement 包中，本书所有范例都会用到它们。`Heading` 类记录着对象的去向信息。这些信息可能包括对象开始运动时的位置、沿某个方向开始运动时的时间、运动的方向以及运动的速度。如果某个对象正处于加速状态，那么它就会包含这些信息。

`Converger` 类是用来执行平滑算法的，它记录了 3 种航向信息。

- `course` —— 对象最新的航向状态。一旦接收了对象的最新信息，这种航向就会及时更新。
- `interceptor` —— 这种航向会使对象在某方向上以某速率运动，以使其在未来短时间内平滑过渡到正确航向。每次 `course` 航向更新后都会重新计算 `interceptor` 航向。
- `view`——这种航向每一帧都会更新，它使用 `interceptor` 或者 `course` 中的值在屏幕上呈现对象的位置。当需要在屏幕上运动那些对象的可视化形象时，程序会使用 `view` 航向而不是 `interceptor` 或者 `course` 航向。

我们使用一个 `Converger` 类实例来定位对象（比如一辆汽车或一枚子弹）。而这就要给运动对象添加一个 `Converger` 类实例。在后面即将介绍的范例中，`Converger` 类实例（以下简称为“converger”）是公开的，每当传入新数据时，游戏就将其更新。每当帧刷新时，converger 的 `run` 方法就会被执行，这个方法会更新 `view` 航向以及对象在实际的航向收敛过程中的位置。

现在让我们来看看两个范例。虽然其用户操作方式大不相同，但是处理平滑过渡的代码却是一致的。

1. 驾驶汽车

本例中你通过移动鼠标来控制一辆汽车，汽车会转着弯开向你当前的鼠标位置。当汽车距离你的鼠标位置不到 50 像素的时候，它就会慢慢地停下来。而当你的鼠标再次远离停止的汽车的时候，它又会加速到最大速度然后跟着你的鼠标运动。



注意　在 book_files/chapter7/car 目录中可以找到这个范例。



注意　这个范例中并没有考虑游戏中正常情况下汽车的转弯半径问题。

汽车的真实航向在每一帧中都会被更新。每过 250 ms，如果汽车位置发生变化，或者它准备加速或减速时，那么汽车的当前航向截屏就会被发送到服务器端。除了正常控制的汽车外，你还会看到一辆渐隐的汽车（图 7-15）。我们把这种渐隐的汽车叫做镜像。就像我们之前介绍的那个基于帧运动的外星人范例一样，这里的镜像代表着其他客户端上可能看到的你的汽车的当前位置。

现在我们来看看如何使用 Converger 类以及其中包含的航向。你能在 Car 类的构造函数中找到下面两行代码：

```
_converger = new Converger();
_converger.interceptTimeMultiplier = 7;
```



提示 你可以试着修改这个值，看看它对实际的运动会有什么样的影响。

创建并存储一个新的 Converger 类的实例，然后用它来更新汽车位置以及平滑过渡路径。第二行则是控制以多快的速度使 interceptor 航向收敛到正确航向（即 course 航向）。数值越大，未来的收敛过程就会越久。当然，数值越大代表着运动更为平滑，不过更新到最新位置就会更慢；数值越小则代表运动过程会更加剧烈，但是能更快地更新到最新位置。通过对本章范例的测试，我发现 5 到 10 之间的值能够在平滑度和精确度之间达到一个很好的平衡。

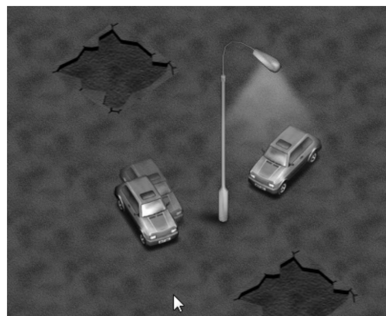


图 7-15

Converger 类的实例需要能够访问程序中所使用的 Colck 类的实例。在 CarExample 类中，在 car 变量创建之后，clock 变量直接设置到了 converger 上。

同样，在 Car 类中你还能找到一个标记为公共的 run 函数。CarExample 类会在每一帧中调用该函数。下面就是 run 函数的内容。

```
_converger.run();

x = _converger.view.x;
y = _converger.view.y;

var rotationIndex:int = NumberUtil.findAngleIndex(_converger.
➔ view.angle, 10);
gotoAndStop(rotationIndex + 1);
```

第一行的代码更新 Converger 类中的 view 航向，然后在 view 航向中找到位置信息并更新汽车位置。后两行代码决定应该显示汽车哪一帧的画面。这辆汽车由 36 张图片组成（每旋转 10° 对应一幅画面）。



注意 在检测对象应该显示哪一帧画面时，NumberUtil.findAngleIndex 方法非常有用，它只需要知道每帧中对象的角度以及角度递增值即可。

CarExample 类负责把你加入到房间，根据在线玩家来添加或者删除汽车，以及发送与接收航向更新消息。我们通过在每一帧中调用 updateHeading 函数来更新汽车的航向。这个函数首先判断是否应该更新汽车航向。如果应该更新，它就会执行下面的代码。

```
checkMousePosition();

var ang_rad:Number = Math.atan2(mouseY - _myCar.y, mouseX -
➔ _myCar.x);
var ang:Number = ang_rad * 180 / Math.PI;

_myCar.run();
var course:Heading = _myCar.converger.course;

if (course.speed > 0) {
  course.angle = ang;
  course.x = _myCar.x;
  course.y = _myCar.y;
  course.time = _clock.time;
}
```

checkMousePosition 函数会检查你的鼠标位置，如果它离汽车足够近就使其减速，如果离汽车足够远就使其加速。接下来的两行代码会计算汽车当前应该转向的角度。

接着程序就会调用你的汽车的 run 方法，以确保在 converger 变更之前其所包含的所有信息都是最新的。一旦汽车发生了运动，我们就可以直接更新 course 航向中的位置与角度信息。值得注意的是，无论何时，只要改变了 course 航向，你都要及时地更新其 time 属性。

当你想要汽车加速或减速时，你可以通过改变 course 航向中的其他一些属性来将其更新。这些都在 accel 和 decel 函数中进行。当希望把航向设置成加速时，你必须更新下面的这些属性。

- time——加速开始时的时间。
- accelTime——加速过程花费的时间。
- endSpeed——当完成了加速后，应该达到的最终速度。
- accel——加速度的值，计算公式为 (speed-endSpeed)/accelTime。

在 accel 和 decel 函数中，我们能够保证汽车至少有 500 ms 的时间来进行加速或减速过程，然后立即发送一个更新消息到服务器。

发送更新消息到服务器的方式与我们之前所看到的范例相同。因此我们只关注那些设置到 EsObject 对象上的属性。在 formatHeading 函数的下列代码中，代表你的汽车当前 course 航向信息被传入 EsObject 对象中。

```
esob.setNumber(PluginConstants.X, heading.x);
esob.setNumber(PluginConstants.Y, heading.y);
esob.setNumber(PluginConstants.SPEED, heading.speed);
esob.setNumber(PluginConstants.ANGLE, heading.angle);
esob.setNumber(PluginConstants.TIME, heading.time);
esob.setNumber(PluginConstants.ACCEL_TIME, heading.accelTime);
esob.setNumber(PluginConstants.END_SPEED, heading.endSpeed);
esob.setString(PluginConstants.NAME, _myCar.playerName);
```

如果汽车没有加速，那么 ACCEL_TIME 和 END_SPEED 都为 0 。将这些属性组合起来就能完整地描述汽车当前的航向信息。

当客户端收到更新消息后，最终都会由 handleUpdateHeading 函数来对它进行处理。这个函数从 EsObject 对象中获取所有的航向信息，然后更新对应的汽车所引用的 converger 属性（只要不是你的汽车）。下面就是用于更新汽车的 converger 属性的代码（所有这些值都来自 EsObject 对象）。

```
var path:Heading = new Heading();
path.x = x;
path.y = y;
path.speed = speed;
path.time = time;
path.angle = angle;
path.accelTime = accelTime;
path.endSpeed = endSpeed;

car.converger.intercept(path);
```

我们先创建一个 Heading 新对象，并为其赋值，然后调用 converger 的 intercept 方法，并将这个 Heading 新对象传递给该方法。intercept 方法接受这个 Heading 新对象，并把它设置为新的 course 航向，然后再创建一个新的 interceptor 航向用于稍后向 course 新航向收敛。游戏运行过程中，不断更新的汽车会从当前运动的车首方向向平滑地过渡到更新后的 course 航向。再看一下图 7-9，你就能对航向拦截的含义有一个更深刻的理解。

2. 移动外星人

在该范例中，你要用方向键控制着一个走动的外星人——这个外星人就是范例“Here I am”中的角色。根据按下的方向键的不同，它总共能够在 8 个方向上运动。当角色开始或停止运动的时候，我们使用一点点的加速度来隐藏网络延时。



注意　在 book_files/chapter7/arrowkey_timesync 目录中可以找到这个范例。



站在多人游戏的立场来讲，这个范例并没有介绍什么新的内容，它只不过再次演示了 Converger 类的用法。Guy 类用于代表每个行走角色。就像之前的范例一样，在构造函数中创建 Converger 类的实例，然后用它来控制角色的运动，如下所示。

```
_converger = new Converger();
_converger.interceptTimeMultiplier = 5;
```

请注意，与前面的汽车范例比起来，这里我们所设置的 interceptTimeMultiplier 属性值要稍小一些。由于角色运动得相对较慢，不会太过偏离其应该在的位置，所以我们能够很快地完成收敛过程。

ArrowKeyExample 类中的代码与之前 CarExample 类中的代码惊人地相似。主要的区别在于捕获用户的输入以及将这些输入映射到你的角色的 course 航向变动上。在汽车范例中，我们利用汽车与鼠标之间的角度及距离来控制一切。而在此范例中，我们每一帧都要检查在 4 个方向键的哪些组合键被按下，然后由此来将 course 航向角度更新为 8 个角度其中之一。这些都是在 moveGuy 函数中完成的。

如果 course 航向发生了变化，那么每 250 ms 就会向服务器发送一个更新消息。如果是加速运动，那么更新消息会立即发送。处理接受更新消息的代码和汽车范例中完全一样。角色将会沿着平滑的路径过渡到正确航向。平滑运动中的角色并不一定会贴着仅有的 8 个方向运动，它会选择最佳的方向沿着一条直线尽快地过渡到正确的航向。

更好的收敛方式

虽然 Converger 类能够很好地将预测路径平滑过渡到更新后的路径从而实现对象位置同步，但我还是认为有一些地方值得改进。

我们现在所使用的 view 航向是使用一个缓冲公式来更新其角度属性的，该公式可将 view 航向角度迅速地变为 interceptor 航向或 course 航向的角度。尽管这样可以实现平滑地旋转，但它并不能做到与对象运动方向保持完全一致；这将导致对象先是停下来，然后再稍微旋转一个角度。另外，我觉得 Converger 类应该完全摒弃较艺术化的旋转。Converger 类能在很多不同情况下重复使用。但你有没有想过，运动对象的旋转过程不单会限于它所在的路径，而且还会与其所处环境有关。不错，一个行走的角色的旋转角度应该会与 view 航向中的旋转角度保持一致。但如果是一辆在冰上行驶的汽车呢？如果是水上的船舶或在小行星带中飞行的飞船又当如何？在这些情况下，物体的旋转过程确实会受到周围环境的影响，而不仅仅限于其所处路径。

另一个需要改进的地方是处理加速的过程。就现在而言，在对象完成加速过程之前，对象都会一直保持在某个方向上。在这些范例当中，这并没有什么问题，因为我们只需要一段很短的时间来完成延迟隐藏。但是假如加速过程本身就是游戏的一部分呢？如果你开发了一个模拟真实环境的赛车游戏，加速的过程需要比较长的时间，而且在这个过程中角度会不断地发生变化，那么在 Converger 类中你就需要一个更好的处理方式。

最后，我觉得收敛器应该吸取一下本章之前所讨论过的另一种处理光滑过渡技术（7.5.2 节中的方案 4）的优点。为了达到更好的效果，这当然需要花一些时间和测试，但是这样做一定会有意想不到的收获。

第 8 章

游戏大厅系统

在创建了多人游戏后，必须考虑如何让玩家加入到游戏中。大厅系统（lobby system）可以帮助我们解决这个问题。大厅系统（有时候会被叫做“游戏大厅”或者就叫做“大厅”）被用来引导玩家加入多人游戏。我们可以用多种不同的方法来建立功能完备并能显示信息的大厅。

本章中我们会了解大厅系统的常见功能并体验一下设计良好的 Flash 多人游戏大厅系统。我们还会为玩家进入或者退出游戏展示一个通用的流程，这里我们把流程的步骤分解成 4 个不同的状态。最后，我们将看到一个专门为本章设计的简单大厅系统。

8.1 常见功能

如上所述，大厅的核心功能就是让玩家加入多人游戏。让我们先来了解一下多数大厅的常见功能，然后再来了解一下那些不怎么常见的功能，如图 8-1 所示。



图 8-1 大厅

聊天——这是一个很常见但非强制性的大厅功能。尽管聊天功能不是让玩家进入游戏的必需条件，但能让玩家谈论游戏内容。他们会因此成为游戏中的竞争对手或被对方添加为好友。不过令人吃惊的是，如此常见而又“受期待”的功能，并不是所有的游戏大厅都会提供。

游戏列表——很多大厅系统允许你查看服务器上的游戏列表。这其中有些游戏已满员且正在进行中，有些未满员的游戏正等待着其他玩家加入，可能有些游戏还设置了密码保护。在客户端上有多种方式能够实现游戏列表功能。如果可以的话，大厅还能显示一些特殊的游戏具体信息，比如像射击游戏正在使用的地图或者当前正在游戏中的玩家名称等。

快速加入——大厅中通常都会有一个快速加入的按钮，这样玩家就可以最快地进入游戏。一旦你单击该按钮，大厅系统的服务器端会搜寻一个开放的游戏并将你添加进去。如果没有的话，它会帮你创建一个新游戏再将你加入其中。

创建游戏——玩家可以通过单击创建游戏按钮来创建一个已经存在的游戏的新副本，而不是加入一个已经存在的游戏。如果玩家希望创建一个新的游戏，那么就要为这个游戏设置一些参数，比如游戏名称、是否需要密码保护，以及游戏特定的参数设定（如允许游戏持续多长的时间）。创建游戏的过程可简可繁，而这完全取决于玩家的需求。

现在让我们来看看那些在大厅系统中很少看到的功能。

挑战系统——这种挑战游戏中其他玩家的功能最可能会出现在回合制游戏中，比如国际象棋，但也不仅限于此类游戏。挑战系统听上去挺简单，但实际上要想正确地实现它却很麻烦（参看下面注释）。

邀请系统——不同于挑战系统的是，邀请系统假定邀请人已处于游戏等待状态（参看下面注释），并且你（邀请人）希望指定的玩家能够加入到你的游戏中。任何玩家都可以邀请一个或多个玩家加入游戏。

复用性——这一功能对于开发人员来说很重要。也许每个负责管理游戏的大厅界面与功能各不相同，但大多数情况下都需要实现相同的基本功能。由于开发大厅系统可能会需要好几周时间，所以就省时而言，你该考虑在前期多花些时间去开发一个通用且易复用的系统，以备在未来项目中尽量予以复用。

挑战中的挑战

为大厅系统或者虚拟世界开发挑战功能是一件挺让人头疼的事。为了让挑战功能运作正常，你得考虑并处理很多问题。这里仅介绍其中一些，希望它们对你将来解决问题有所帮助。

先来看一种理想状态吧。玩家 A 挑战玩家 B，玩家 B 的屏幕上显示出一个是否接受挑战的对话框，然后他点击了“接受”。接着玩家 A 收到了接受挑战的消息，两个玩家很快进入游戏对战起来。嗯，不算太坏——就这么简单。

不过有些偶然性事件却会破坏这种理想状态。例如，玩家 A 挑战玩家 B，玩家 B 屏幕中显示出对话框，但玩家 B 并不打算回应它。这就得解决两个问题。首先，在玩家 A 的屏幕上应该能显示一段表示挑战某人的动画。这段动画应该能够被取消或者因为某种原因被终止；其次，在没有作出回应前玩家 B 应该能一直看到对话框。如果添加一项功能允许玩家 A 取消挑战的话，那么就必须通知玩家 B 挑战已取消，这样才能去除玩家 B 所没有回应的对话框。另外，如果此次挑战可以被取消，并且在玩家 A 取消的同时玩家 B 最终却接受了挑战，那么玩家 A 尽管收到了这个回应，也不得不认定此次挑战无效。只有把这些问题都考虑清楚，你才能处理好这种情况。

下面是另外一种情况。玩家 A 向玩家 B 发起了挑战，玩家 B 的屏幕上显示一个是否接受挑战的对话框。然后玩家 C 又向玩家 A 发起了挑战，当然玩家 A 的屏幕上也会出现一个相同的对话框。假如玩家 A 后来接受了玩家 C 的挑战，那么玩家 B 就需要知道这些情况，然后移除玩家 A 向自己发起挑战的对话框。但是假如玩家 A 接受玩家 C 挑战的同时，玩家 B 也同时接受玩家 A 的挑战呢？如果不能很好地处理这种情况就会出现严重错误。因此，或许可以设定当玩家 A 还存在没有处理的挑战时，他就自动拒绝玩家 C 的请求。

在挑战系统中还会出现更多的棘手情况，比方说你想挑战的玩家突然离线等。我并非想让你对开发挑战系统深怀恐惧而踯足不前，而只是想让你对将来要面对的问题有个思想准备。最好的解决办法就是由服务器端处理逻辑。那会非常有用。

8.2 游戏流程

考虑一下当玩家从大厅系统进入游戏时，你希望带给他们什么样的用户体验？也许不用想你就能说出下面的流程：

- 选择游戏列表中你想玩的游戏；
- 看到该游戏加载的画面；
- 游戏开始；
- 游戏中；
- 游戏结束；
- 返回游戏大厅。

这就是玩家完成全部用户体验的步骤，大厅主界面在其间既是起点又是终点。在此过程中会有些基本状态发生改变，我们把这个过程叫做游戏流程。

上面我们说的流程都是显而易见的，但还有另一种同样很普遍的游戏流程，它更多是从开发者角度来描述的并且适用于所有类型的游戏。下面每一项都是玩家将会存在并维持的状态：

- 等待状态；
- 初始化状态；
- 游戏进行状态；
- 游戏结束。

以上描述的每个状态都比你想的要复杂一些。现在我们就针对每个状态作更详尽地阐述。



注意　具有很明确的开始与结束状态的典型多人游戏适合采用这种流程。假如是合作模式的游戏，即玩家可以在任何时候加入和离开游戏而游戏本身不会结束，那么另拟一个游戏流程会比较合理。

8.2.1　等待状态

当玩家想玩游戏的时候，他可以创建一个新游戏、加入一个已存在的游戏，或者选择快速加入游戏。当服务器端收到这个请求后，假设他所希望加入的游戏还没开始，也没有满员，那么他就可以加入其中，并进入等待状态。

等待状态就是指一个或多个玩家等待更多玩家加入与最终游戏设定生效的状态。在此状态中你能看到一个新界面，其中显示了一些空余栏位以待新玩家加入。当玩家加入时也就填充了这些空栏。如果游戏还允许玩家使用更多自定义设置，那么他们还可以做得更多，比如说可以一起来表决下一赛程玩哪张地图。

某些游戏还要求玩家将其就绪状态标识出来。尽管玩家加入游戏并已处于等待状态，但他们还是要将状态标识为就绪。这样做的理由在于，如果是那种机器人对战游戏，那么每一个玩家可能都会需要在开战前定制他的机器人。

大多数游戏都有一个能接受玩家数量的范围（例如，一个纸牌游戏需要 2 ～ 4 个玩家）。当进入等待状态并把自己的状态标记为就绪的玩家数达到了游戏最少需求人数后，倒计时开始（图 8-2）。倒计时的目的是给那些还未标明就绪的玩家一个机会让其标明状态。当然，在倒计时过程中也允许更多玩家加入进来。



注意　对于简单的游戏来说，所有这一切都可迅速自动完成。玩家只要一进入等待状态就会自动被标记为“就绪”，这就能避免使用所有可能的复杂界面，虽然使用它们也能达成目的。

在游戏倒计时的过程中，假如玩家数量回落到小于游戏设定的最小值，那么倒计时将会停止直到条件重新满足。当倒计时完成时，那些被标记为就绪的玩家将进入初始化状态，而未被标记为就绪的玩家将被送回游戏大厅。



图 8-2 倒计时过程中的等待状态



注意 有些游戏的大厅主界面中包含了等待界面。

8.2.2 初始化状态

对于即将开始的游戏，这个状态让每个客户端有足够的时间来创建用户初始化数据。例如，在等待状态中玩家们都定制了各自选用的机器人，并选择了战场地图。所有客户端都必须加载这些数据，然后游戏内容才能在屏幕上显现。

在此过程中，很多游戏唯一要做的就是告诉服务器端初始化结束。考虑一下国际象棋游戏：已没有什么要加载或者初始化的了（假设主要的游戏文件已被预先加载进来了）。

假如游戏确实需要加载一些数据，这个过程就可以通过一个进度条来显示给每个玩家。用这种方式，所有玩家都可以看到对手的加载进度（图 8-3）。

一旦所有玩家都通知服务器初始化完成并且各自就绪，游戏就开始了，所有玩家进入游戏状态。

8.2.3 游戏进行状态

如你所知，进入此状态后，玩家已各就各位而且也加载并建立了所需的一切（图 8-4）。现在游戏就能开始了！



图 8-3　游戏加载

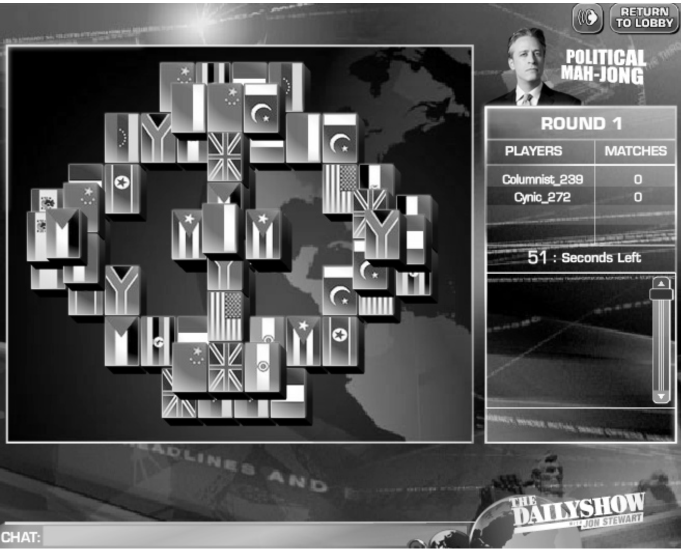


图 8-4　游戏过程中

这个状态是专门用于执行自定义游戏逻辑的。在特定的游戏结束条件被满足前，一切皆有可能。当游戏结束后，玩家就会进入游戏结束状态。

8.2.4　游戏结束

虽然游戏业已结束，但在此状态下玩家们依然聚在一起。通常游戏结果会在此时显示给所

有玩家。玩家们可能还会聊聊刚才玩游戏的情况。

你可以对你的系统进行编程以便让玩家从此状态返回到等待的房间或者回到大厅（如图 8-5 所示）。



图 8-5　游戏结束

8.3　游戏：挖宝 2

为了演示游戏大厅的最基本的功能，我们依然以第 6 章的挖宝游戏为例，对它进行改进（稍后讨论），然后创建一个大厅系统以帮助玩家查找与加入游戏。我们还将介绍 ElectroSever 中有助于创建大厅系统的内置功能。

8.3.1　全新的ElectroServer概念

ElectroServer 提供了一些非常有助于开发大厅系统的功能。插件可以作为游戏（我们称之为服务器游戏）注册到 ElectroServer 上。每个服务器游戏都有一个唯一的字符串 ID，我们把它称作游戏类型。通过使用游戏类型并调用一些新的 API，客户端就可以完成以下任务：

- ❒ 加载包含所有某类型游戏的列表；
- ❒ 快速加入一个某类型游戏；
- ❒ 加入一个已存在的特定类型游戏；
- ❒ 创建一个新的某类型游戏。



注意　服务器上的游戏只不过是一个与单个或多个插件相关联的房间而已。

在客户端，只有当你发送了请求游戏列表的消息后才会用到 `ServerGame` 类。该类包含了下列游戏相关信息。

- **游戏类型**——用来表明游戏类型的字符串名称。
- **游戏 ID**——每个游戏类型所拥有的唯一 ID 号，用于加入指定类型的游戏。
- **锁定游戏**——游戏控制插件能够锁定游戏。游戏被锁定后就不再允许新玩家加入。通常游戏未开始时是非锁定状态，而当玩家满员或者游戏开始后就会变成锁定状态。
- **游戏详情**——这个属性包含一个 EsObject 对象，对游戏而言它是可选的。其目的是发布游戏自定义信息。假如玩家们在房间里正等待更多玩家加入。他们选择了一张下一回合要玩的新地图。更新游戏详情时就能包含这些信息，然后你就可以用多种方式（图标、数字、视频等）将这些信息最终显示在大厅上。
- 当讲到后面的范例时我们会介绍如何使用下面的 API 调用。但对初学者而言，在开发服务器游戏时你会用到下面这 4 个请求和 1 个响应事件。
- FindGamesRequest/FindGamesResponse——它们被客户端用来加载某类型游戏的列表。如果想使游戏列表保持最新，你得每过 10 s 或 20 s 重新加载一次才行。响应消息中就包含着列表。
- CreateGameRequest——用来创建指定类型的游戏。在此请求过程中，自定义游戏详情会被发送给服务器端的游戏插件。
- QuickJoinGameRequest ——当玩家希望随意加入一个游戏时就会用到它。将一个游戏类型名称作为过滤条件，然后服务器就会检索该类型游戏的列表以查找可玩的游戏。通常服务器会将玩家加入到第一个可玩的游戏中，或为玩家创建一个新游戏。
- JoinGameRequest——当玩家确定了所希望加入的游戏类型后（兴许就是游戏列表中的某一个），我们就会用到该请求；在发出该请求时会添加游戏 ID。
- CreateOrJoinGameResponse——玩家可以通过刚提到的 3 种方式进入游戏：创建游戏、直接加入、快速加入。最终，对这 3 种请求的任何一个，其响应都是 `CreateOrJoinGameResponse`。该响应表明了加入成功还是失败。如果失败，它会包含一个错误信息，如果成功，则它会包含所加入房间的信息。

8.3.2 大厅系统范例

我们对第 6 章中的挖宝游戏进行了少量修改，现在你可以在游戏中看到所有玩家的鼠标状态，比如鼠标运动以及挖宝的过程；游戏开始时会有一个倒计时，这时更多玩家还可以加入；当某个玩家的得分达到 10 000 点时游戏结束。



注意 你可以在 book_files/chapter8/lobby_system 文件夹下找到相关源文件。

此外，我们还为游戏专门创建了一个大厅系统（图 8-6），以方便玩家进入游戏。这个大厅

系统相当简单，它拥有如下的功能：

- 聊天；
- 游戏列表；
- 加入列表中的某个游戏；
- 快速加入。

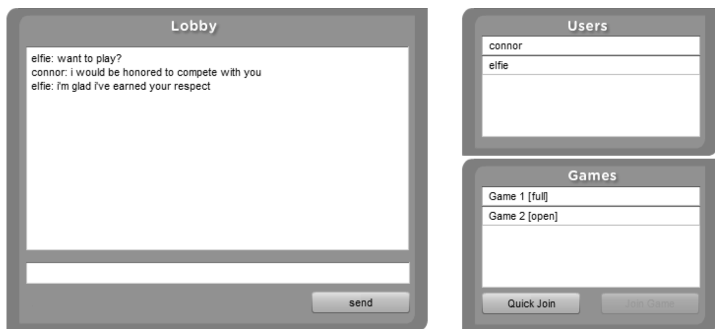


图 8-6 修改后的挖宝游戏的大厅系统

接着让我们来看一下新引入的 ElectroServer 请求和响应接口以及如何使用它们。然后我们将了解如何处理游戏流程中的 4 个状态：等待状态、初始化状态、游戏状态和游戏结束状态。

1. 请求与响应

让我们来看看这个简单大厅系统是如何使用所有刚讲过的新请求的（除了 CreateGame-Request 不是新的外）。为了在界面上显示游戏列表并始终保持其为最新状态，我们需要每 2 秒钟向服务器请求一次列表信息。这是通过一个定时器来实现的。当执行定时器时就会调用 Lobby 类中的 onGameListRefreshTimer 函数：

```
//create request
var fgr:FindGamesRequest = new FindGamesRequest();

//create search criteria that filters the game list
var criteria:SearchCriteria = new SearchCriteria();
criteria.setGameType(PluginConstants.GAME_NAME);

//add the search criteria to the request
fgr.setSearchCriteria(criteria);

//send it
_es.send(fgr);
```

我们先创建一个新的 FindGamesRequest 请求对象，再为其传入一个 SearchCriteria 类的实例（它还能用于更高级的过滤功能，但那就超出了本书范围）。就目前而言，我们只想为它提供一个游戏类型名称以返回该类型所有游戏的列表就足够了。当客户端加载完列表后就会

调用 onFindGamesResponse 函数。传递给该函数的响应事件对象中含有一个 ServerGame 类的对象数组，只要把该数组传递给 refreshGameList 函数就能使游戏列表中的数据显示出来。游戏列表是通过一个列表框组件来显示的。所以接着我们遍历游戏列表，并把每个游戏对象都格式化为一个显示在列表框中的元素。下面就是遍历逻辑中的一段代码：

```
var game:ServerGame = games[i];
var label:String = "Game " + game.getGameId();
label += " [" + (game.getLocked() ? "full" : "open") + "]";
dp.addItem( { label:label, data:game } );
```

我们先是引用了上述迭代中的 ServerGame 对象数组元素，然后我们用它们来格式化那些要提供给列表框组件的数据项。label 属性就是列表框中的数据项标签，而 data 属性就是每个数据项所存储的数据内容。用来显示游戏列表中对象的标签是由“Game”和游戏 ID 所组成的，其后还要加上“[open]”或“[full]”标记。我们由 getlocked 属性值来判断某个游戏是公开状态还是满员状态。

当玩家单击列表框中的一个游戏后，Join Game 按钮就会变成可用状态。单击它之后，将创建一个 JoinGameRequest 对象并将其发送给服务器端。为此我们需要调用 joinGame 函数并为其传入一个 SeverGame 类的实例。下面就是该函数的内容：

```
private function joinGame(serverGame:ServerGame):void {
  var jgr:JoinGameRequest = new JoinGameRequest();
  jgr.setGameId(serverGame.getGameId());
  jgr.setGameType(PluginConstants.GAME_NAME);

  _es.send(jgr);
}
```

首先创建了一个 JoinGameRequest 对象实例，然后将用于指定具体游戏的游戏 ID 以及游戏类型添加进去。接着将该请求发送给服务器端。接下来我们要介绍快速加入的功能以及处理响应的代码。无论是哪种请求，客户端都会收到相同的响应。

在游戏列表框下面有一个 Quick Join（快速加入）的按钮。如前所述，快速加入可以帮助玩家加入一个公开的游戏，或者当没有公开游戏时会为其创建一个新游戏。点击快速加入按钮就会执行下列代码：

```
private function quickJoin():void {
  _quickJoinGame.enabled = false;

  var qjr:QuickJoinGameRequest = new QuickJoinGameRequest();
  qjr.setGameType(PluginConstants.GAME_NAME);
  qjr.setZoneName("GameZone");
  _es.send(qjr);
}
```

首先我们看到 Quick Join 按钮被禁用了。这是因为，发送一个快速加入请求就能确保玩家最终会加入游戏（即便要为此创建一个新游戏），所以我们要禁用这个按钮，这也是为了避免在请求过程中按钮被再次单击。创建请求对象并在其中设置游戏类型名称。当然我们还

设置了区名称，这样就可以在该区中创建游戏房间。然后将请求发送，最终我们会收到一个 CreateOrJoinGameResponse 事件。

这个事件由 onCreateOrJoinGameResponse 函数处理。该事件对象包含一个 success 属性。如果玩家成功加入游戏，那么 success 属性为 true，否则就为 false。如果玩家加入游戏失败，那么该事件对象中会包括一个错误描述。如果成功加入游戏，该事件对象会包含玩家所在房间 ID 和区域 ID 的信息。这些信息需要被记录下来，因为在游戏过程中客户端需要知道消息应该往哪里发送。

当大厅检测到玩家已经加入游戏并存储了房间 ID 和区域 ID 后，它就会发布 JOINED_GAME 事件，接着 LobbyFlow 类就会捕获到该事件。然后 LobbyFlow 类移除了游戏大厅，并通过传递相关 API 引用以及房间信息从而初始化 DigGame 类的实例。

2. 游戏流程的状态

在我们的大厅系统范例中，我们总用经历了 4 个状态，但是只有 3 个是可见的。其中初始化状态是静默处理的。当玩家收到游戏加入成功的 CreateOrJoinGameResponse 事件时，玩家还处于等待状态。然后在客户端，我们移除大厅界面，创建游戏界面，也就是创建了一个 DigGame 类的实例。

ElectroServer API 的一个引用与房间信息被传递给 DigGame 类。然后该类立即发送一个行为类型为 INIT_ME 的插件消息。当服务器收到此消息后就知道客户端已经做好了准备，可以开始游戏了。在这种情况下，虽然当前玩家已经初始化完成，但游戏仍处于等待状态。

如果当前游戏中只有一位玩家，那么在等待状态中将会显示“Waiting for players...”（等待玩家）的文本消息（图 8-7）。一旦第二位玩家加入其中，10 s 的倒计时就会启动并显示出来。在倒计时的过程中，其他玩家也可以加入。假如在此过程中，玩家数量又重新回落到 1，倒计时将会立刻停止并重新显示“Waiting for players...”的文本消息。任何在倒计时过程中加入的玩家，都能看到计时正确的倒计时。

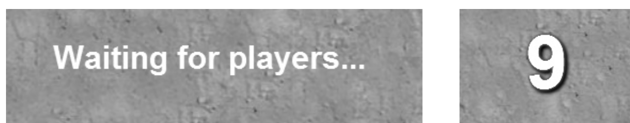


图 8-7 等待玩家

倒计时结束后，所有玩家都已把自己标记为已初始化状态，游戏就结束了初始化状态而进入游戏状态。玩法跟第 6 章中一样，唯一区别在于只有得分达到 10 000 点时游戏才会结束。

当某个玩家的得分达到 10 000 点后，游戏就会进入结束状态。在这个状态中，会显示一个根据玩家得分多少排序的玩家列表。另外，还提供了一个可以让玩家在任何时候回到大厅界面的按钮（见图 8-8）。

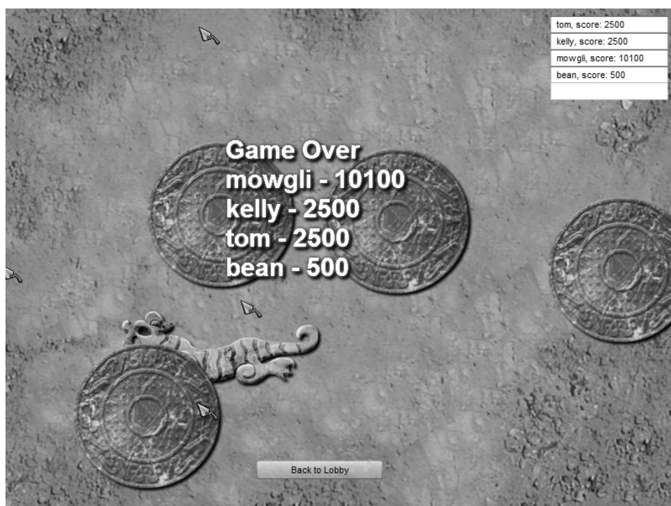


图　8-8

这个大厅系统还可以加以改进，比如允许刚刚结束游戏的玩家们再玩一次，这需要将他们送回等待状态而不是大厅界面。

8.3.3　在ElectroServer中注册游戏类型

当 ElectroServer 启动时，每个游戏类型都要通过游戏管理器（Game Manager）注册。通常，你只需要一个服务器端的插件就能一次性为所有游戏注册。



注意　我们的扩展插件可以在 www.electrotank.com/gamebook/server/src/com/gamebook/gamemanager/GMSInitializer.java 找到。

使用 GMSInitializer 插件中的 `initOneGame` 方法，你可以很轻松地注册游戏，只要该游戏没有默认的细节设置，而且游戏名称跟插件名称一样。一个游戏可以关联多个插件，但是大多数游戏只会关联一个。如果你还有另外一些游戏也需要游戏管理器来扩展的话，只需要在 `GMSInitializer` 类的初始化方法中，针对每个游戏都调用一次 `initOneGame` 方法。当然你也可以为你的一个或多个游戏编写自定义的方法，从而代替默认的方法。

接下来，打开 ElectroServer 的管理面板，并跳转到 Extensions 标签。单击加号（+）旁边 GameBook 标签。如果在服务器级别的组件中没有 GMSInitializer，那么单击按钮把它作为一个新的服务器级别的组件添加到列表中。你可以使用任何唯一的字符串作为插件的名字，当 ElectroServer 启动时，这些插件也会随着运行。重新启动 ElectroServer，然后检查控制台或 ElectroServer4.log 文件。你应该能看到一条警告级别的记录文本行，显示的是“DiggingPlugin2 game registered with GameManager”。

第 9 章

实时坦克游戏

迄今为止，我们已经了解很多东西。我们知道了使用 ElectroServer API 进行 socket 通信的基本知识，并探究了在游戏中进行客户端—服务器端验证，而且还详细地了解了实时运动的基本概念。在本章中，我们将会把这些概念结合起来完成一个实时多人的坦克游戏。

本章首先将集中讨论如何把迄今为止我们所学过的知识应用于该游戏中，随后还会详细讨论视线的概念，以及如何将它们用于路径验证和判定物体被炮弹击中。此外，我们还将简要地探讨一下关卡编辑器并引入立体声的概念。

9.1 游戏简介

在这个游戏中，你要用鼠标和键盘控制一辆坦克。游戏采用的是顶视角模式（见图 9-1）。当你移动鼠标时，坦克的炮塔就会旋转到你鼠标指针的方向。单击鼠标，坦克就会发射一枚炮弹。如果按住 Shift 键单击鼠标，那么你就会给坦克建立一个新的目标点，它就会朝着该目标运动。

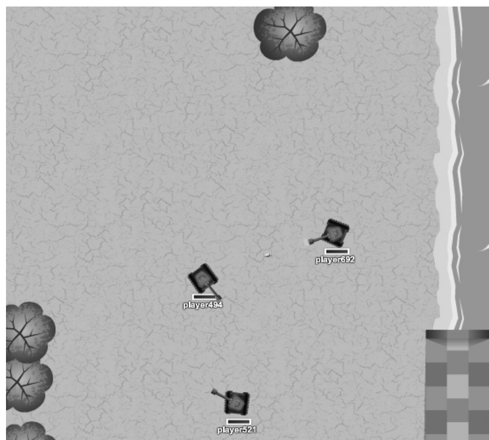


图 9-1



注意 你可以在 book_files/chapter9/tank_game 目录中找到该游戏的源文件。

游戏地图包含很多游戏物件（图 9-2）。有些允许你的坦克通过，有些则不允许。有些会阻止炮弹穿过，有些则不会。这里列出了游戏中所有的物件，以及它们的特征描述。

- 树木——坦克可以从它下方经过而不会被阻挡。当然炮弹也可以穿过。可作藏身之用。
- 房屋——坦克和炮弹都不能通过。
- 水域——坦克不能通过，但是炮弹可以。
- 桥梁——坦克可以通过，炮弹也可以穿过。
- 围墙——既不允许坦克通过，也会拦截炮弹。
- 能量块——游戏中的特殊图标，坦克驶过即可将其拾取。

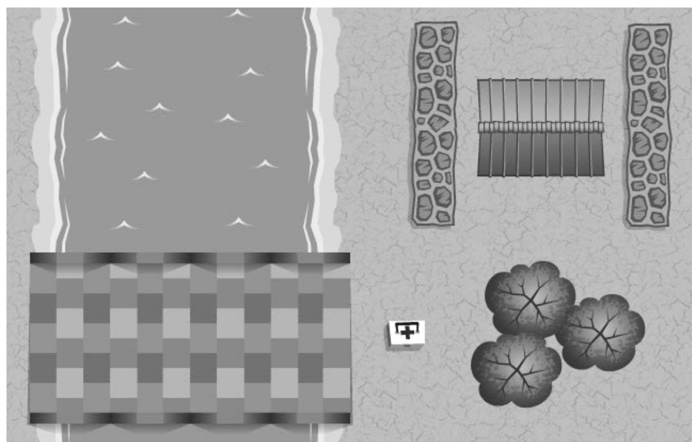


图　9-2

游戏目的就是摧毁对手的坦克。你要做的是向它们开火。游戏本身没有结束条件。如果一辆坦克被摧毁，几秒钟后它就会在另一个地方重生。

游戏中也存在着能量块。坦克通过拾取能量块就能得以优化。游戏中只有一种能量快：生命能量块。当坦克驶过生命能量块时就能将它拾取。每拾取一个生命能量块，坦克生命值就会增加。

游戏的地图尺寸为 1600×1600 像素，而游戏是在 800×600 像素的 Flash 窗口中运行的。因为地图尺寸四倍于 Flash 窗口尺寸，所以地图会随着坦克的运动而卷动。我们会使用一个迷你地图来展示一个缩小版本的完整地图，用圆点来代表坦克。当坦克运动的时候，对应的圆点也会跟着运动。这可以让所有玩家都能实时地看到其他玩家在哪里。

游戏并不只限于一种地图布局。地图布局数据存储在一个 XML 文件中，我们可以用关卡编辑器来创建此 XML 文件。关卡编辑器可以让你发挥创意并创建自己的坦克游戏关卡。在本章的后面将会讨论如何使用关卡编辑器。

9.2 权威和预测

在第 6 章中我们比较了客户端验证与服务器端验证。第 7 章则专门探讨了多人游戏中的实时运动和网络延时隐藏的概念。那些概念构成了本章这个游戏的逻辑核心。我们将在本节讨论该在哪里作出验证，`Converger` 类应该应用在什么地方，并且还会提到针对该游戏的其他一些预测环节。

9.2.1 坦克路径

关于游戏中坦克行驶的路径，有两点值得我们去探讨：首先，坦克如何确定路径；其次，路径确定后，坦克如何沿着该路径运动。

通过在坦克当前位置和鼠标单击位置之间画一条直线，我们就能确定一条路径。在这个游戏中，我们用视线来确认路径是否有效。如果你在坦克当前位置与其预定目标位置间画了一条直线，而某个能阻截住坦克的游戏物件与该直线相交，那么只好否定该路径。为了得到一条有效路径，我们会进行两层检测。当你单击鼠标时，客户端会使用视线检测以确认路径是否会穿过任何地图上的静态物件。如果被认为是一条有效路径，那么该路径信息将会发送给服务器端进行二次验证。

如果路径还要在服务器端被验证，那我们为什么还要不厌其烦地在客户端验证它呢？这是为了让客户端可以立即对用户的输入作出反馈，当然这里的前提是客户端所作的验证必须是正确的。一旦玩家单击鼠标，并且随之形成的路径得到了客户端的验证，玩家的坦克就开始沿该路径运动（在接收到服务器端响应之前）。但假如服务器认为该路径无效，那么客户端就需要修正坦克的运动。我们的目标就是要尽量减少客户端发送错误路径的机会（本章后面部分将会讨论此游戏中有关视线的代码）。

第二个值得讨论的问题是坦克如何沿路径运动。这个坦克游戏使用了我们在第 7 章中讨论过的 `Converger` 类。由于坦克的运动速度相当地缓慢，事实上在它正常停止之前，我们就能够确认它最终的目标位置，所以游戏中我们并不需要靠加速度让坦克平滑地停下来。这里唯一出现的新内容就是 `Heading` 类中的 `targetX` 和 `targetY` 属性。我们用这些属性来指定坦克的最终目的地。

9.2.2 射击

当玩家移动鼠标时，坦克炮塔会旋转到指向鼠标的方向。炮塔的旋转与坦克的运动路径无关。玩家单击鼠标一次，坦克就会沿炮塔朝向发射一枚炮弹。

当玩家单击鼠标朝坦克射击时，客户端会发送一条消息到服务器端。只有当服务器端判定此次射击有效时，客户端才能发射炮弹。据我测试，这样做导致的服务器延迟并不明显，所以

用户体验还算不错。不过，如果希望反应更迅速，那么直接让客户端发射炮弹就行了，就好像它已得到服务器端的响应一样。这是有可能的，但这样做也会引出一些问题。主要的问题在于，每颗炮弹都有一个唯一的 ID，这使服务器在炮弹射出之后可以查找到它，当它击中某物体时，服务器会将此消息通知客户端，或者在某些情况下服务器端会通知客户端将其移除。如果客户端在得到服务器端响应之前就已发射出了炮弹，那么这颗炮弹就没有 ID，它以后也就无法被服务器端引用。所以你必须跟踪这颗没有 ID 的炮弹，并且当接收到服务器端确认后再为其设置一个 ID。另外，当服务器端通知客户端此次射击无效时，你还必须移除这颗炮弹。

我们使用 `Converger` 类来控制炮弹的运动。炮弹在某个特定方向上以固定速率运动。炮弹从坦克炮口射出的位置与客户端收到响应消息时炮弹所在的位置之间会有一段间距，这个间距可由 `Converger` 类中的网络延时隐藏算法轻松解决掉。

9.2.3　碰撞检测

碰撞检测用来确定一个对象是否与其他对象发生了碰撞。在我们这个游戏中，它指的是要检测炮弹是否与坦克或者静态物件发生了碰撞。碰撞是否发生最终是由服务器端来判定的。不过，由于炮弹的速度很快，所以为了产生更好的用户体验，我们可以让客户端对炮弹将与何物碰撞做出一些预测。这是为什么呢？首先让我们来理解这个问题。这类似于第 7 章中简略提到过的有关赛车不减速就突然停止的问题，即如果不采取平滑修正或预测运动措施，通常就会出现渲染误差。本游戏中的炮弹每秒运动 240 像素。设想一下炮弹朝坦克方向前进。当服务器端检测到发生一个碰撞时，它就会发送一个事件给所有客户端，用来通知有碰撞发生（图 9-3）。如果客户端收到消息的延迟是 50 ms(这个延时还是比较低的)，那炮弹就会多运动 12 像素。这还是最好的情况。如果延迟再略高一点或者碰巧延迟出现了一小段激增，那么这个差距就只会更大，而这将导致在整个游戏过程中始终存在这样的视觉偏差。由于这种问题在游戏中很常见，所以我们有必要找到一种解决方法。



图　9-3

问题现在已很明确了，为了尽量减小因碰撞事件接收的延时所造成的影响，我们来了解一下这个游戏所采取的措施。炮弹在游戏中的“死亡”方式有 3 种：炮弹达到了射程限制、炮弹击中了静态障碍物（如围墙），以及炮弹击中了坦克。下面我们依次对它们进行讲解。

本游戏中炮弹的射程大约有 600 像素，速率为每毫秒 0.24 像素，这也就是说一颗炮弹的生存期最多约为 2 500 ms（见图 9-4）。当炮弹发射后，我们把炮弹因到达射程而被移除的那个时

刻值存储起来。在炮弹发射后，它在每一帧中的位置会根据其经过的总体时间而得到更新。如果当前帧的时钟时间大于或等于该炮弹的自然“死亡时刻”，那么我们就移除这颗炮弹。如果在炮弹生存期完结时还没有发生任何碰撞的话，那么我们也应该将它移除。针对服务器端响应的不同可能性，你必须持保留态度。例如，子弹在其生存期的最后时刻可能会击中坦克，但无论出现任何情况，你都该知道此时是炮弹的“死亡时刻”。



图　9-4

只需稍加思考，你就能发现预测炮弹与静态障碍物的碰撞是比较简单的。炮弹以固定的速率沿一个指定方向运动。而障碍物是不会运动的。发射炮弹时，我们就会查看炮弹离开炮口时所处位置以及炮弹到达射程时所处的位置。根据这两点画一条直线，这条直线就是炮弹可能的运行轨迹（图 9-5）。如果该直线与任何障碍物发生了相交，那就表明炮弹在到达射程之前就会与这些障碍物中的某一个发生碰撞。然后我们沿着这条路径检测出所有可能的碰撞点，然后找到离炮弹的发射点最近的可能碰撞点。我们将这个可能发生的碰撞时刻存储下来。如果游戏运行时间已经到达了我们预测到的碰撞时间，而炮弹仍然存在，我们就将它移除（有关用视线实现碰撞检测的内容将在 9.3 节中介绍）。

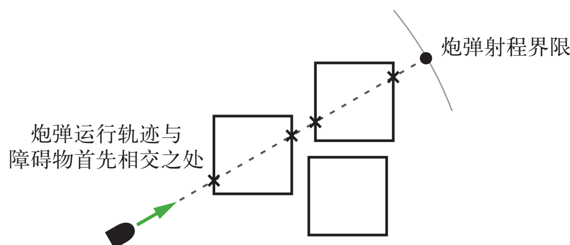


图　9-5

迄今为止，我们已了解了如何预测炮弹死亡的 3 种情况中的前两种。最后一种情况是预测炮弹与坦克的碰撞。对此我们可以使用两种方法：基于帧的碰撞检测或者预先碰撞检测。本章所选用的方法较简单但不太精确。预测炮弹与坦克碰撞的最简单的方法是逐帧检测当时是否发生了可见碰撞。由于使用的是时间同步，所以如果你看到在客户端有碰撞发生，那么这极可能就是一次真的碰撞，只是你还没有收到服务器端的响应消息。只要客户端检测到碰撞在此刻确实发生，它就会阻止炮弹前进。除非收到服务器端认同的响应，否则我们将不会播放炮弹爆炸动画或声效，也不会降低坦克生命值。

此方法容易出现的问题是，它只是看到了当时的一个截屏画面。这有可能（甚至很有可能）炮弹与坦克确实发生了碰撞，但是我们无法用这种技术检测到，因为我们只会在指定时刻查看炮弹的位置。



注意　我们可以使用这种并非十全十美的客户端碰撞检测方法。其结果还算不错。因为服务器端会检测所有的碰撞，所以最坏的情况也不过是在客户端得知发生碰撞之前，炮弹已经飞过目标一小段距离。

还有另一种预测碰撞的方法——这可能是客户端用到的最准确的方法了。在任何指定时刻，我们都知道所有坦克与炮弹的轨迹。假如没有服务器的更多响应，我们也能够百分之百地准确预测出所有可能会发生的碰撞，并且我们用来检测两个运动物件间碰撞的方法与帧无关。只要给出两个物件的路径，这种方法就可以确定这两个物件是否曾经碰撞过以及（如果发生碰撞）碰撞的时间。据此方法，每辆坦克都有一个针对每颗炮弹所产生的碰撞列表。该列表按时间进行排序以确定哪个碰撞最先发生。不过，你也有可能会根据过时的位置信息（坦克的）预测出发生了碰撞。因而在这种游戏中，你永远不能完全相信碰撞预测，因为这些碰撞结果是根据完全无法预测的物件运动得出的！所以还是尽你所能使用现有的信息来做好碰撞预测吧。另外，我想告诉你的是，你根本找不到关于上面我们说的这种特殊方法的代码范例。

9.3　视线

视线检测就是从某个位置开始沿着某个方向延伸出一条直线，直到该直线触碰到某物体为止。视线在游戏中有很多用途。下面是其中的几个。

- 它可确定 AI 生物的视域。这样一来，它们就不会对那些看不到的事物作出反应。
- 在射击游戏中它能确定子弹如果射出后将击中哪里，或者只确定运动的子弹即将击中哪里。
- 它可用于验证选择的路径是否有效。

我们使用视线来验证游戏中的坦克想要行进的路径。另外，如果炮弹首先击中的不是坦克，我们还会用视线来预测炮弹会击中何种静态物件。下面让我们来了解下如何计算视线，以及如何将它用于路径验证和碰撞预测。

9.3.1　线段交点

这个游戏中，子弹从某个位置开始运动，如果没有击中任何东西，那么它终结的位置是可预测的。在子弹起点和终点间画一条直线就构成了线段。当我们为坦克创建一条新路径时，它的起点和终点同样也构成了一条线段。所有的静态物件（比如围墙）都是矩形，所以静态物件边界也是由四条线段组成的。以此角度来考量该游戏，你就会明白如果能找到一种方法检测出线段间的交点，那我们就能确定坦克的行进路径是否有效或者炮弹会跟哪个物体发生碰撞。

我们使用 com.gamebook.utils.geom 包中的一些类来表示线段并检测线段间的交点。在演示怎样使用这些类之前，让我们先来简单介绍下其中的数学原理。

任何直线都可以用方程式 $y = mx + b$ 来表示（见图 9-6 及表 9-1）。

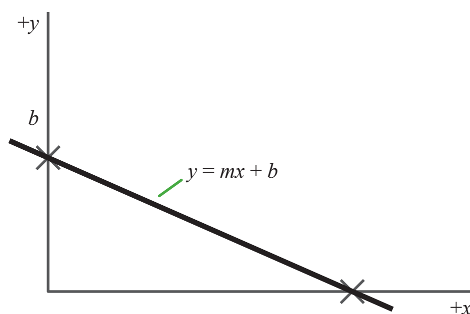


图　9-6

表 9-1　方程式 $y = mx + b$ 中各符号代表的意义

符　　号	意　　义
m	直线的斜率，斜率越大，直线越陡
b	直线与 Y 轴交点的 y 坐标值（如果直线能与 Y 轴相交）
x	x 坐标值
y	输入的 x 坐标值所对应的 y 坐标值

接着考虑我们面对的情况，假如你有两条直线，你就可以这样表示它们：

$$y_1 = m_1 x_1 + b_1$$
$$y_2 = m_2 x_2 + b_2$$

如果两条直线相交，那么在交点上它们有相同的 x 与 y 坐标值。因此，上述两个方程式在该交点处可用如下等式表示：

$$m_1 x + b_1 = m_2 x + b_2$$

然后求解 x 值，得到

$$x = (b_2 - b_1) / (m_1 - m_2)$$

这就是两条相交线交点的 x 坐标值。把这个 x 值代入到以上任何一个方程中，你都能求得该交点的 y 坐标值。

仅仅因为两条直线相交并不能说明两条线段也能相交。找到两条直线相交以后，我们还需要进一步确认该交点是否同时都存在于两个线段上。如果存在的话，那么线段就是相交的（见图 9-7）。

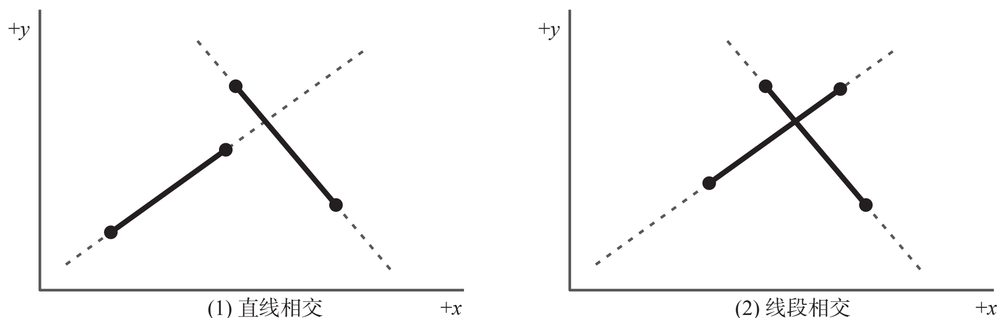


图　9-7

现在，我们来看看用来处理这些逻辑的相关类。LineSegment 类用来表示一条单独的线段。你可以像下面这样创建它并对它赋值：

```
var seg:LineSegment = new LineSegment(new Point(10, 20),
➔ new Point(15, 30);
```

Point 类是 ActionScript 类库中的内置类。它只存储 x 与 y 坐标值。LineSegment-Collection 类存储了一个 LineSegment 类的对象数组。它有助于将构成形状的线段进行排序。你可以像这样创建它：

```
var seg1:LineSegment = new LineSegment(new Point(10, 20),
➔ new Point(15, 25));
var seg2:LineSegment = new LineSegment(new Point(5, 15),
➔ new Point(20, 30));
var collection:LineSegmentCollection =
➔ new LineSegmentCollection();
collection.addLineSegment(seg1);
collection.addLineSegment(seg2);
```

然后我们使用一个 IntersectionDetector 类来与以上两种类共同检测直线交点。该类公开了用于检测的两个静态方法。其中，检测任意两线段间交点使用的是 segmentSegmentTest 方法，检测线段与线段集合间交点所使用的是 segmentCollectionTest 方法。这些方法都会返回一个 IntersectionTestResult 类的实例。该实例包含一个表示交点已测出的属性，以及一个描述该交点位置的 point 对象。

segmentCollectionTest 方法可以发现多个交点。如果你正在测试一个路径是否有效，那么你所需要的全部信息就是这些交点。但是如果用该方法来检测炮弹何时会击中某物体，那么你就得知道哪个交点离炮弹来源位置最近了。该方法允许调入一个可选的 point 对象参数用于距离的检测。如果提供了这个 point 对象参数，那么所返回的碰撞点将会是与调入的点距离最近的点。

9.3.2　路径验证

Map 类是通过解析地图布局数据来创建关卡的。同时它也为地图中各种各样的静态物件存

储了所需的所有线段信息。TankGame 类包含了游戏中的大多数逻辑。当玩家按住 Shift 键单击鼠标时，TankGame 类就试着为坦克创建一条新的路径。以下是执行函数的部分代码：

```
private function sendNewWayPoint():void {
  var course:Heading = _tankManager.myTank.converger.course;
  var view:Heading = _tankManager.myTank.converger.view;

  //坦克当前位置
  var sx:int = view.x;
  var sy:int = view.y;

  //坦克目标位置
  var tx:int = _map.mouseX;
  var ty:int = _map.mouseY;

  //检测路径有效性
  if (_map.validatePath(new Point(sx, sy), new Point(tx, ty))){
```

开头两行代码创建了对客户端坦克的 course 与 view 航向的引用。接着后面的 4 行代码存储了坦克当前位置及其预期目标位置的 x 与 y 坐标值。这些坐标值被传递给 Map 类的 validatePath 方法。如果该方法返回 true，那就说明该路径有效，并且代码还会向服务器发送一条消息（服务器还要对此进行验证）。以下是 Map 类的 validatePath 方法的前几行代码：

```
//创建一条连接两点的线段
var seg:LineSegment = new LineSegment(point1, point2);

//检测线段与所有障碍物之间会发生的碰撞
var res:IntersectionTestResult = IntersectionDetector.
➥ segmentCollectionTest(seg, _pathItemsCollection, point1);
```

首先我们根据传递进来的起点和终点创建一条线段。然后使用 IntersectionDetector 类检测该线段是否与构成地图上所有静态障碍物的线段集合相交。地图上每个物件都有一个名为 obstacle 的布尔值属性，如果其为 true，则不允许坦克通过；另外还有一个名为 hittable 的布尔值属性，如果其为 true，则不允许炮弹通过。在上面的代码中，我们对路径与所有障碍物进行一一验证，排除掉其中会碰撞到的物件。注意到 point1 变量作为第 3 个参数传递给了 segmentCollectionTest 方法。这就是说，如果检测到有直线交点，我们就能知道哪个交点离 point1 最近。

如果路径验证失败，那么我们就播放一段声效，并且还会在玩家坦克所处位置与路径和障碍物发生碰撞的位置之间画一条直线，通知玩家此路不通。放心，这条直线会很快会消失的。

9.3.3 碰撞预测

正如本章前面所讨论的那样，预测一枚炮弹会在何处与静态对象发生碰撞是很简单的。客户端一旦收到射击事件，不管会不会发生碰撞，它都会执行一个检查，看看碰撞会在什么地方

发生。我们把碰撞点以及碰撞时刻保存下来。如果炮弹应当与障碍物发生碰撞的时刻已到，那我们就把炮弹从屏幕上清除掉。

以下是 TankGame 类中 parseAndApplyBulletHeading 方法内部的一小段代码：

```
var life:Number = 2500;
var endx:Number = course.x + course.xspeed * life;
var endy:Number = course.y + course.yspeed * life;

var point:Point = _map.getCollisionPoint(new Point(course.x,
➥ course.y), new Point(endx, endy));
```

你能计算出炮弹沿其路径所到达的极限位置。炮弹起点与终点位置被传递给 Map 类的 getCollisionPoint 方法。该方法会把一条线段跟所有可击中物件进行交点检测，然后返回第一个交点。以下是该方法中的所有代码：

```
var seg:LineSegment = new LineSegment(point1, point2);
var res:IntersectionTestResult = IntersectionDetector.
➥ segmentCollectionTest(seg, _bulletItemsCollection, point1);
var point:Point = res.point;
return point;
```

根据子弹运行的起点与终点创建一条线段。然后把该线段与所有可击中对象进行比较以检测交点，并返回距离 point1 最近的交点。如未发生碰撞，返回值就为 null。

9.4　游戏消息

正如本书多次提到的那样，客户端与服务器端的通信是通过在两者间传递格式化的并且带有 ACTION 变量的 EsObject 对象来实现的。该变量值指明了消息的用途以及该对象所包含的其他数据。在这里我们不打算把所有消息对象的格式都完整地定义出来，而只是列出了 ACTION 变量的名称以及它们的作用。

表 9-2　EsObject 对象的 ACTION 变量与它们的作用

ACTION	发送方向	作用描述
INIT_ME	客户端到服务器端	当玩家成功加入房间并准备接受游戏数据的时候发送到服务器端。服务器端以布告板状态作为响应
BOARD_STATE	服务器端到客户端	包含了新加入玩家绘制游戏画面所需的所有内容——地图数据、屏幕上能量块的种类、所有坦克和炮弹的位置以及它们的目标
ADD_TANK	服务器端到客户端	当有新坦克加入时发送给所有客户端
REMOVE_TANK	服务器端到客户端	当有坦克离开时发送给所有客户端
UPDATE_TURRET_ROTATION	客户端到服务器端 / 服务器端到客户端	包含了玩家坦克炮塔的旋转信息；客户端周期性地将该消息发送到服务器端。除非把其他玩家的炮塔旋转信息集成在一起，否则服务器不会做任何处理，接着集成消息被返回给所有客户端。查看 9.6 节以了解其详细信息

（续）

ACTION	发送方向	作用描述
HEADING_UPDATE	客户端到服务器端 / 服务器端到客户端	当玩家按住 Shift 键单击以确定一个新路点的时候，客户端会先对它进行验证，然后发往服务器端。服务器紧接着对其验证，如果有效则返回给所有客户端。如果无效，客户端就会收到包含服务器为该坦克处理的最后一次有效航向的错误消息。客户端用它来修正路径
SHOOT	客户端到服务器端 / 服务器端到客户端	当玩家单击开火的时候发送到服务器。为了不让客户端开火速度过快，服务器会对其进行验证，然后发送一个开火消息给所有的客户端
SHOT_HIT	服务器端到客户端	服务器会跟踪所有坦克和子弹的位置，并检查它们之间碰撞的可能。如果检测到某个坦克被击中了，服务器会通知所有客户端
HEALTH_UPDATE	服务器端到客户端	发送给所有的客户端，以通知它们某个坦克的健康值发生了变化 (当坦克被击中，它的健康值就会发生变化。如果健康值为 0，坦克就被摧毁了)。我们使用一个单独的消息，而不是将信息包含在 SHOT_HIT 消息中，其原因就是为了像介绍的那样以一种新的方式获得伤害值信息
TANK_KILLED	服务器端到客户端	当某个坦克的健康值为 0（并且被摧毁了）的时候，发送给所有客户端。坦克会在另外一个地方立即重生，也就是 SPAWN_TANK 消息的内容
SPAWN_TANK	服务器端到客户端	当坦克被摧毁并在新位置获得重生的时候，该消息就会发送给所有的客户端
COLLECT_POWERUP	客户端到服务器端 / 服务器端到客户端	当坦克发现能量块时，该消息被发送给服务器端。服务器会对其进行验证，看看坦克是否离能量块足够近来拾起能量块，然后通知给所有客户端
SPAWN_POWERUP	服务器端到客户端	通知所有客户端在屏幕上增加一个新的能量块 (你应该还记得，被拾起的能量块 10 s 后会在同一个地方重新出现)
ERROR	服务器端到客户端	一旦发生错误就会发送给客户端，例如客户端射速太快或者客户端正尝试行进一个无效的路径

9.5 迷你地图

在游戏中，地图尺寸通常是屏幕尺寸的很多倍。所以一般屏幕上会有一个完整区域的微缩表现形式——迷你地图。迷你地图的主要目的是让玩家感知其在虚拟世界中所处的位置，在很多情况下，其他玩家也会出现在虚拟世界中。

迷你地图通常会以某种方法定义其风格，而不是像字面意思那样只是一个真实地图的微缩版本。主要建筑物可以用图标来表示，那些动态运动的对象（比如玩家）则可以用彩色点或者其他适当的游戏小图标来表示。迷你地图的表现方式可以是丰富多彩的，也可以简单到只用单一色块来表示围墙和玩家。

在坦克游戏中，我们采用一种方便编程的方式来创建迷你地图。在地图被创建好但还没有添加坦克时，我们创建了一个 BitmapData 实例，然后使用 draw 方法把地图绘制到位图数据中。接着我们创建一个 MiniMap 类的实例，并把 BitmapData 实例传递进去。BitmapData 会被添加到 Bitmap 对象中，其尺寸缩小到当前大小的 1/16，接着被添加到舞台上。并且我们还给它添加了阴影。

现在，我们已经有了一个较大地图的微型版（见图 9-8）。从图像风格来看，很容易就能看出这个地图是实际大小的 1/16。我们不一定非得用更有创意的方法来显示地图信息。一旦玩家加入游戏，MiniMap 类中的 addTank 方法就会被调用。它将会创建一个点来表示坦克，并将其添加到屏幕上。所有点的位置逐帧都会被更新，而该位置是根据相应坦克真实位置的 1/16 来确定的。

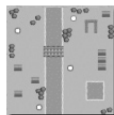


图　9-8

最终，所有坦克的位置都会在迷你地图上显示出来，并且还会实时更新。玩家自己的坦克显示为一个带有绿色光晕的白点，而对手的坦克则显示为带有蓝色光晕的白点。

9.6　消息集成

利用 socket 服务器来处理游戏消息会非常好。它能以事件驱动机制进行多人游戏编程。如果碰撞发生，服务器端会通知客户端。如果客户端进行了某种操作，例如更新了航向，它也会让服务器端知道。但如果消息的传送频率过高，那么为每个事件都创建一条独立的消息就会造成一些麻烦。从 Flash Player 中我观察到一个现象，随着消息接受频率的增加，客户端会感觉到延迟越来越大。消息能按时到达，不过 Flash Player 还需要花费一点时间对它们进行处理，以便提交给 ActionScript 使用。

很难确切地说每秒钟客户端限制处理多少消息才算合适，不过我尽量保持客户端每秒处理 5 个或更少的消息。当消息接受频率激增时，客户端有可能会接受到数倍的消息，例如刚好发生了一些碰撞，或者一些新玩家刚加入游戏。这些情况都还能应付得来，但是在游戏过程中，你要尽可能地保持低频率的消息传输。

相对而言，在这个坦克游戏中，射击动作频率不算太高（大概平均每秒一次），运动频率也不高（大概每 10 s 设置一次新路径）。不过旋转炮塔的频率却非常高。炮塔总是要立刻面向鼠标的方向。客户端每秒会向服务器端发送两次炮塔旋转更新的消息，以便其他客户端也知道炮塔的更新。如果游戏中有 10 辆坦克，而且要确保更新消息能发送到所有客户端，那么单单为了更新炮塔信息，我们就要每秒创建 20 个客户端消息。这可太麻烦了。



注意　作为客户端开发人员，你并不需要知道消息是否是集成消息。ElectroServer API 接收到集成消息后会把它重新拆分成结构化的更小的事件个体。然后这些事件个体作为独立的 PluginMessageEvent 事件被触发。如果你收到一个由 5 个消息组成的集成消息，那么将会有 5 个 PluginMessageEvent 事件被触发。

解决方案是消息集成，即把多个消息收集在一起然后放到一个较大的消息包中发送，而不是发送单个消息。ElectroServer 的插件可选择将某些消息集成起来一起发送，而不集成另外一些消息。（例如，在收到射击和碰撞事件之前，我们不愿意再为此多等 50 ms。）在这个坦克游戏中，我们把所有的炮塔更新消息集成在一起，然后每秒发送两次。无论游戏中有多少坦克，你都不会在 1 秒钟内收到超过两次的炮塔更新事件。这使得 Flash 能很快地处理这些消息然后交付代码执行。

9.7 关卡编辑器

关卡编辑器，或是地图编辑器——无论你叫它什么，对于大多数游戏和虚拟世界而言，你都需要这么一种工具。关卡编辑器（level editor）可以让你用一种高度可视化的方式创建自己的关卡（或称地图）。相对于直接在记事本里编辑 XML 数据，这可是很棒的方法。在开发期间，使用一个关卡编辑器创建布局，这对于快速查找和修复地图相关的 bug 是很有帮助的。完成开发之后，你还可以用它不断地为你的游戏创建新关卡。在这一节中，我们将会讨论关卡编辑器的作用，以及如何来存储关卡数据。



注意 此游戏的关卡编辑器的 Flash Develop 项目文件是 Tank Game Editor.as3proj。

关卡编辑器使用了大量与游戏中代码相同的代码。启动编译过的编辑器程序后，你会立刻看到两个按钮：open（打开地图）和 new map（新建地图）（图 9-9）。点击 open 按钮，你可以看到一个对话框，该对话框允许你浏览本地磁盘中已经存在的地图文件（地图文件以 XML 作为后缀）。如果你选择了一个有效的地图文件，它就会被载入，然后显示地图。如果你点击 new map 按钮，一张新地图将会被创建，但此时文件还没有保存。



图 9-9

创建了一张新地图后，你可以看到只包含地面的画面、save（保存）按钮、一个可以输入保存文件名的文字段，还有一些可以被拖到虚拟世界中以代表相关物件的小图标，如图 9-10 所示。



图 9-10

单击 save 按钮后，你会看到一个让你选择文件存储位置的对话框。当应用程序试图保存文件的时候，它会收集地图信息并将其格式化到一个 EsObject 对象中。EsObject 对象有一个

`toXML` 的方法，该方法把一个 XML 格式的字符串存储到文件中。当文件将来被载入时，一个新 EsObject 对象就会被创建，并且 XML 对象会被传递给 `fromXML` 方法以还原 EsObject 对象。然后应用程序使用 EsObject 对象的属性重建地图布局。

创建地图是非常简单的。通过把屏幕上方的对象简单地拖拽到地图上就可以创建一个新对象。任何时候你都可以拖动地图上的任何对象。单击并拖到一个新位置，然后释放它。双击它，就可以移除对象。场景大小是不可编辑的，同样地面纹理也不行。不过，为了丰富地貌，你可以往地面上放置水域对象。

当你把鼠标放到距离程序上下左右边界 10 像素以内时，屏幕就会卷动。

你可以像添加普通游戏物件那样为游戏添加坦克重生点和能量块重生点。它们指定了能量块的位置以及坦克被击毁后的重生之处。

这个坦克游戏关卡编辑器会使让你的工作变得很轻松。不过如果你打算将它应用到一个更大的项目，它依然有很大的改进余地。我马上想到了以下主意：

- 允许改变地面材质的纹理；
- 允许玩家设置地图的大小；
- 使用一种更直观的方式卷动地图；
- 把游戏对象添加到带有滚动条的列表中，这样就可以为编辑器增加更多的游戏对象。

9.8　立体音效

音效往往是小游戏中最容易被忽视的部分。我目睹过无数的小游戏产品，开发持续了 4 ～ 8 个星期，而直到最后的几天项目才开始设置（或者创建）音效。精妙的音效不仅会使游戏添色不少，还能整体提高用户体验。设计得当的音效通常不会让人察觉到——它只会使用户沉浸于游戏之中从而增强用户与游戏的沟通。

除了选择合适的音乐和创建合适的音效以外，你可能想知道如何通过音效来增强用户的游戏体验。那么你就应该考虑立体声。

立体声的作用就是根据声源相对于玩家的位置调整声音的效果。在 Flash 中，我们可以通过设置音量（volume）和偏移（pan）来控制声音。把这些应用到坦克游戏中，可以让我们根据距离的远近听到炮弹爆炸的不同声音，还能根据声音事件的水平位置产生相对于左右扬声器的不同的偏移效果。

在坦克游戏中，我们根据所有立体声事件相对于屏幕上可视区域的中心位置来调整它们的音量大小（见图 9-11）。如果一事件发生在 200 像素以内，我们将保持其音量为最大值。除此之外，随着像素的增加，我们逐渐减小音量，直到距离超过 400 像素，音量就会降为 0。音效偏移量将会以 400 像素为左右极限值而设置在 −1 到 1 之间（见图 9-12）。

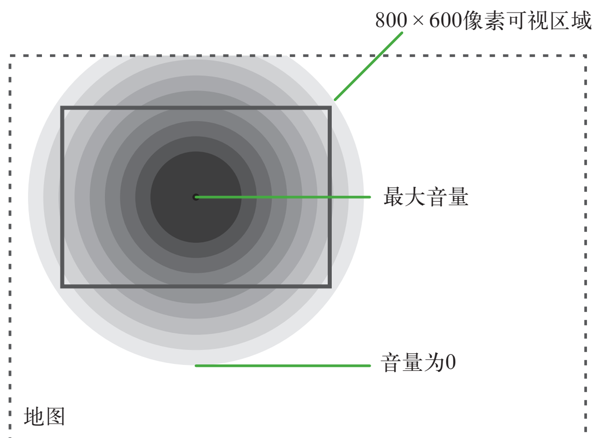


图 9-11

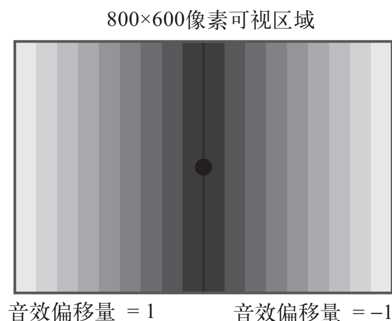


图 9-12

让我们来看看应用到坦克开火音效的代码。下面是在 TankGame 类的 parseAndApply-BulletHeading 方法中的代码片段：

```
var snd:Sound = new ShootSound() as Sound;
var transform:SoundTransform = getSpatialSoundTransform(.35,
➔ new Point(course.x, course.y));
if (transform.volume > 0) {
  snd.play(0, 0, transform);
}
```

这样，一个播放声音的 Sound 类实例就被创建出来了。然后，我们调用 getSpatial-SoundTransform 方法，它会接受一个你所希望播放声音的最大音量参数和一个该声音事件位置参数。然后返回一个 SoundTransform 对象，该对象已经设置了正确的音量大小和偏移值。如果音量仍然大于 0，那么我们就播放声音。

getSpatialSoundTransform 方法中对声音的音量和偏移计算进行了相关处理。自上而下，我们把这个函数分解为 4 段进行讲解。下面是第 1 段：

```
private function getSpatialSoundTransform(maxVolume:Number,
➔ point2:Point):SoundTransform {
  //center of screen
  var point1:Point = new Point( -_map.x + _viewWidth /
  ➔ 2, -_map.y + _viewHeight / 2);
```

我们希望使用的最大音量值设置被传递进来。音量大小将会在 0 和这个最大值之间变化。声音事件发生的地点通过参数 `point2` 传递进来。另外还创建了一个名为 `point1` 的变量来存储地图上可视区域的中心位置。

接着往下看第 2 段：

```
var volDis:Number = Math.sqrt(Math.pow(point1.y - point2.y, 2)
➔ + Math.pow(point1.x - point2.x, 2));
var maxVolDis:Number = 400;
var volumeMultiplier:Number = 1 - Math.min(Math.max(volDis -
➔ 200, 0) / maxVolDis, 1);
```

变量 `volDis` 存储了声音事件与地图可视部分中心位置之间的距离。变量 `maxVolDis` 存储了自最大音量范围 200 像素以外的声音逐渐消失的范围值。最后一行则创建了一个介于 0 到 1 之间的归一化数值。1 是最大音量，0 是最小音量。通常它会介于这两者之间。

下面的第 3 段代码用来处理偏移量计算：

```
var maxPanDis:Number = 400;
var panMultiplier:Number = (point2.x - point1.x) / maxPanDis;
panMultiplier = Math.max( -1, panMultiplier);
panMultiplier = Math.min(1, panMultiplier);
```

变量 `maxPanDis` 存储了我们将偏移值设置为 −1 到 1 之间的水平最大距离。我们创建了一个归一化因子 `panMultiplier` 并将其值控制保持在 −1 到 1 之间。如果声音事件产生的位置位于距离地图可视区域中心点左边 400 像素或更远地方的话，将完全用左扬声器播放声音；如果其距离位于右边 400 像素或更远地方的话，将完全用右扬声器播放声音。而中间的任何地方，将会在两个扬声器之间设置均衡值，声音会在左右扬声器间变化。

该函数的最后一行如下：

```
return new SoundTransform(maxVolume * volumeMultiplier,
➔ panMultiplier);
```

我们创建了一个 `SoundTransform` 对象并将其返回。音量和偏移的值根据归一化数值的计算结果得以设置。音量大小是根据传递进来的最大音量值和变量 `volumeMultiplier` 的值的乘积计算而来。

立体声（或者只是在游戏中动态地去调整音量）确实能够让用户体验更具现场感。你可以试着在简单的游戏（如桌球）中使用动态音量。不应该让所有的碰撞声音都相同。在这样的游戏中，你最好根据动量转移去调整音量。在虚拟世界里，同样可以使用这种在坦克游戏中讨论过的技术。实在没必要让几乎处于画面外的声效和更接近你的焦点区域的声效相同。

第 10 章

区块式游戏

区块技术很早就应用于电子游戏中了。大家可能都玩过区块式游戏，其中过去很受欢迎的有《大金刚》（Donkey Kong）、《食豆小子》（Pac-Man），还有《赛尔达传说》（Legend of Zelda）系列等。区块是一种被放置在游戏中并被重用（reused）以创建整个游戏地图的视觉组块。对于游戏开发者与策划人来说，使用区块技术有很多好处。

你可以在本章暂时先不关注多人游戏。第 11 章中的游戏以及后续几章所讨论的虚拟世界都将用到我们在本章中所讲解的概念。本章我们将着重概述区块式游戏的概念以及它们为游戏所带来的性能表现与游戏逻辑上的提升。由于虚拟世界（包括本书中的“古老家园”）中广泛采用了 A* 算法，所以本章我们还将对它予以详述。

10.1 区块式关卡与绘制式关卡

如今，多数大型商业游戏都是 3D 类的，这些游戏大量使用 3D 模型来创建实时渲染的场景和人物。而当处理平台为 Flash 或者很多手持设备时，它们不具备足够的性能去渲染每个 3D 模型，因而你需要另辟蹊径来创建游戏世界里的关卡或场景。我们权且把它们叫做关卡吧。除了使用 3D 模型外，我们还有两种主要的创建关卡的方法：绘制式的和区块式的。

绘制式关卡是一种只靠美工人员绘制或者其中的游戏物件是由手工摆放的关卡类型。（第 9 章的坦克游戏就是这种方式的范例。）接着，开发者必须找到一种自定义方式来编写游戏策略以使其能作用于被交付的美术作品。假如使用了大量自定义代码来确保关卡以期望形式去运作，那就可能会非常耗时。在坦克大战游戏中，我们就做了一些假设来简化必要的代码。

区块式关卡要比绘制式关卡更加结构化。区块式关卡将许多独立区块有序地组织起来，从而构成一个完整的关卡场景。在采用侧视角或顶视角的游戏中，关卡场景经常配置在沿屏幕 x 轴与 y 轴延伸排列的网格中，就像《塞尔达传说》那样。绝大多数区块式虚拟世界都采用菱形区块（将在第 12 章予以讨论），如图 10-1 所示。

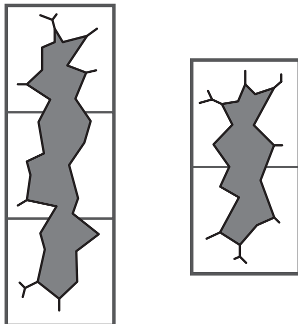


图 10-1



注意　有些游戏采用的是六边形区块，这虽然罕见，但的确存在。

你也可以在关卡中混合使用区块式与绘制式方法，使用单独的大图来做背景，然后选择使用区块来完成有逻辑性的数据存储、角色定位以及碰撞检测。本书后面会讲到的虚拟世界就使用了这种混合方法，虽然它还采用了一些可重用物件（如树、岩石以及栅栏）。

一些游戏支持区块层级技术。假如你有个青草层，便可以在其顶部再放置许多花朵区块。层叠的区块要使用 alpha（透明）通道来使上下层很好地混合。观察下面的“Precious Girls Club”（可爱宝贝俱乐部）界面（图 10-2），你就会发现在其青草层上有许多花朵区块。



图　10-2

区块式游戏有诸多优势，关卡设计即是其一。通常我们可以很轻松地为区块式游戏创建关卡编辑器。尽管可能会花大量时间来为其添加一些高级功能，比如说使玩家能够复制并粘贴区块组或者能够在地图格子上“绘制”区块，但基本的关卡编辑代码在游戏自身中却是通用的。

10.2　区块式方法的其他优点

本节我们将讨论区块如何有助于提升游戏程序性能以及如何使一些游戏逻辑更易于实现。

10.2.1　性能

在有些方面使用区块式实现会带来高水平的游戏性能。区块式技术能轻松地处理游戏世界中的碰撞检测，实现高性能的地图卷屏，保持内存低占用率，并且它还能够方便地存储数据。

在我们了解各个性能优势之前，我们需要介绍一个数学技巧来确定任一指定点触碰的是哪个区块。考虑图 10-3 所示的这个 4×4 区块方格，每一个区块的高与宽都为 40 像素。

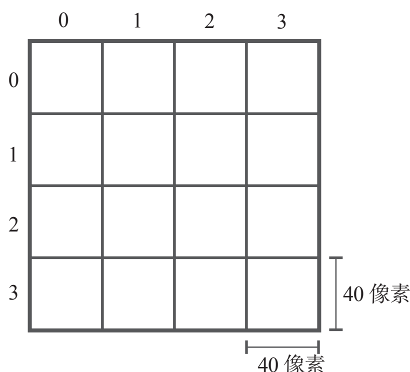


图 10-3



注意 区块式虚拟世界的所有区块尺寸相同。

当游戏中的角色、物件和敌人在虚拟世界中四处运动时，你需要定位他们正经过的区块，这能协助你完成碰撞检测与实现游戏逻辑。那么，在虚拟世界中给定一个点，你怎样确定它在哪个区块上呢？这很简单——只需找出该点所在区块所对应的行列序数就行了（见图 10-4）。

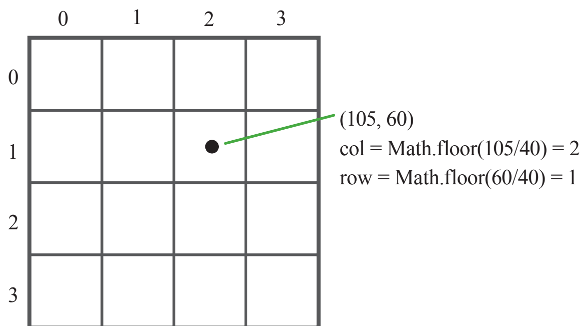


图 10-4

通过下面这条语句来确定该点所在区块的列序数：

```
var column:int = Math.floor(x/tileWidth);
```

而该点所在区块的行序数为

```
var row:int = Math.floor(y/tileHeight);
```

知道了该点所在区块的行列序数，我们就可以访问该点所在区块了。

现在让我们继续看一下区块式方法的其他性能优点。我们在说明的同时也会与非区块式方法对比。

1. 碰撞检测

碰撞检测用来检测一个点何时与一个对象碰撞或两个对象间何时发生碰撞。它可以用于确定某个物品（如钥匙或宝石）何时应该被拾取，或者用于确定一枚炮弹是否击中一个物件。在非区块式游戏里，检测这些碰撞通常会用距离检查。如果你有个非区块式游戏，里面有 1 000 个硬币可以收集。那么每一次游戏循环中需要做 1 000 个距离检查，而这样付出的代价就会很高。

但是如果你在区块式游戏里有 1 000 个硬币，那么碰撞检查就会很简单。

(1) 获取你当前所在的区块。
(2) 检查该区块是否包含硬币。如果有，就收集。

这个范例显示了碰撞检测可以简单到如此地步，而它带给游戏的负面影响则微乎其微。



注意　这并非总是处理碰撞检测的最佳方法。例如，你不可能用这种方法处理类似台球间的精确碰撞。

2. 卷屏

大部分区块式游戏所要显示的内容都比单个屏幕要大。屏幕会随着主角的运动而卷动以保证角色处在视野中。如果地图很大（比如说是屏幕尺寸的 20 倍），那么你能想象得到这会对性能产生多大的影响。而一个非区块式游戏会将那么大尺寸的关卡内容同时全部渲染出来，即使无论何时你都只能看到一个屏幕那么大范围的内容。并且当屏幕卷动时，Flash 播放器还会运动所有的可视元素。

而在区块式游戏中，如不采用措施，你也会遇到类似的要解决的性能问题。区块式游戏的一个优点就在于你拥有一个条理分明的可视化数据网格。当区块进入或者离开可视范围时，它们就会被显示列表添加进来或者移除出去。如果你用这种办法处理卷屏，你就不会再渲染出超过一个屏幕的区块了。

有时程序员会将该方法进一步深化，他们会在屏幕上只使用一个位图实例。所有区块也都是位图且都绘制在这个主要的屏幕位图之上。当关卡卷屏时，你可以采用某种技巧使像素沿卷屏方向滑动，同时新的像素则被添加到另一端。如果使用得当，这将是一项非常高效的技术。

3. 内存消耗

区块式游戏是由在关卡中重复出现的可视元素所组成的。一个关卡可能有 10 000 个区块，而这些区块可能由大约 100 个可重用且独立的形象单元所组成。如果处理得当，游戏每次所使用的区块都是唯一的，而且会使用同样的 `bitmapData` 类。这就使得程序能保持低内存占用率，因为 `bitmapData` 类是共享资源。而在某些游戏中，每个物件和事件都是独立的，这就毫无优势可言，它们只会占用更多内存。

4. 数据存储

通常在区块式游戏里，每一个独立区块都有一个类文件。你可以很方便地将区块相关信息储存到这种类中。它可以存储要显示的图像、该区块上的物品（如硬币或其他可拾取的物品）以及一般的区块属性。一个典型区块属性是 `isWalkable`。如果 `isWalkable` 值为 `true`，那么角色可以出现在该区块上；否则，角色就不能经过该区块。另外一个常见属性是 `isHittable`。如果为 `true`，那么发射的炮弹会停在该区块上；否则，它将穿过该区块。



注意　可存储到区块里的信息没有任何类型限制。就不同游戏而论，区块既可以用作敌人的产生点、下一关入口，也可以只用来容纳可拾取物品。

10.2.2　何时执行游戏逻辑判断

当你拾取到一枚硬币时，程序如何得知这一情况？我们先前讨论过检测碰撞，但没说明应该何时检测碰撞。通常我们是在角色到达区块正中央时开始检测的。在该点检测会使程序在每轮游戏循环中减少一些工作量，然后在适当时检查游戏逻辑。



注意　尽管此种方法有助于检测物品采集、切换开关等逻辑，但当角色已经触碰到区块中央时再检测该角色是否被允许进入区块，这显然不太合适。应该在角色行走前就进行检测。

一种折中的方法

很多区块式游戏的区块与你所控制的主角尺寸差不多大，而且主角只允许停留在区块中央。如果你的角色在一个区块中央正准备向右走，那么它所运动的最小距离就是自原地到右边这一个区块中央的距离。尽管游戏并不都是这样来控制角色运动的，但这种处理方法很常见。这里所讲的区块式的一些优点就是基于这种方法而产生的。

当物件触碰到区块中央时才运行游戏逻辑，这种方法同样有助于我们处理另一方面的问题，即敌人的人工智能（Enemy AI）。虽然有无数种可实现 AI 的编程方法，但其实可归纳为以下两种：跟随预定路线或者执行寻路算法。这两种方法共同需要面对的唯一问题是，当物件触碰到它已经走向的区块的中央之时，其下一步走向哪个区块。

此外，我想讨论的关于区块式方法在逻辑判断上的最后一个优点就是寻路了。但你要知道，角色必须止步于一个区块中心和此优点毫无关系。在一个组织有序的区块网格布局中，我们可以清晰地寻找两个区块之间的路径。如果是为了要让角色沿着此路径运动，那么 A* 算法（这是 10.3 节的主题）就是一个不错的选择。如果我们寻找的是视线路径（例如为了旅行或射击），

那么 Bresenham 直线算法是个很好的选择。Bresenham 直线算法允许你给定起始区块和终止区块，它会返回两个区块之间连线经过的所有区块。本书并没有涉及这个算法，但是它值得你到别处去多了解一下。

10.3　A* 寻路算法

在一些绘制式（非区块式）的虚拟世界中，比如“Club Penguin”（企鹅俱乐部）或“Gaia Online”（盖亚在线），只需朝用户鼠标点击的任何位置走直线即可实现寻路。你可能会立即得出结论：“这用的不就是视线嘛！”——但却不是。这条线可能会经过一个障碍物，然而化身（avatar，此概念将在第 13 章讲到，意即游戏角色）却会沿该直线一直走下去直到遇上障碍物才会停下来。这时候用户不得不用一些更短的路径使化身能绕过障碍物并继续沿着原定的路径走下去。

如果你想在虚拟世界中如此寻路，那么你会很高兴地发现这实现起来确实简单极了。但我个人对这种方法很失望。还有一个更高明的使用区块的寻路术——A*。A* 是一种用来找寻系统从初始状态到目标状态代价最低的算法。（我们很快会谈到什么是代价。）



注意　尽管 A* 算法并不只用于寻路，但是在这里我们只将它用于寻路，这就是我们为什么在本章以及本书余下章节中将其称作寻路算法的原因。

我们来看看该算法的一些特点以及描述该算法的伪码，最后我们来查看一个范例。

10.3.1　算法概念

对于在区块式虚拟世界中用于寻路而言，A* 算法会根据起始区块与目标区块来检测网格以便确定连接起始点和最终点的最低代价区块路径。我们会在后面介绍“代价”的含义，在大多数情况下，代价最低路径指的就是距离最短路径。

该方法首先检测起始区块周围的区块，然后智能地沿一个最有可能成为最终路径部分的方向展开搜索（见图 10-5）。这是 A* 算法与其他寻路算法的一个主要差异：其他许多算法不会在任何特殊方向展开搜索；当到达目标点后它们就会停止搜索。

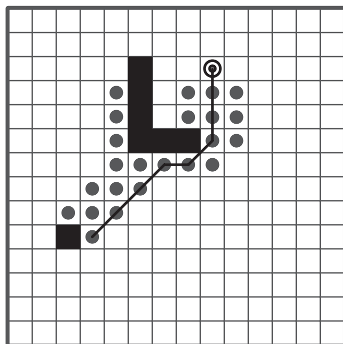


图　10-5

另外，A* 算法还可以考虑地形所带来的影响。这简直太酷了！虽然人们很少用到这一点，不过你要知道，如果要通过一些不理想的区块，那么最短路径就未必是最好的路径。马上我们就会深入了解这一点。

与很多技术概念一样，A* 算法也有几个术语，你应该熟悉它们以便理解接下来我们对算法的阐述。这些术语描述了伴

随寻路过程而产生的状态、行为以及结果。

- 节点（node）就是当前你正在检测的区块。
- 展开（expanding）节点指的是（在代码中）你访问节点的每一个邻近节点。
- 在 A* 里，启发式预估值（heuristic，下文我们统一简称为 *h* 值）是一种基于选定的测量单位所作出的有把握的猜测值。（相当含糊，不是吗？放心，我们很快就会对它有更多的了解的。）
- 代价（cost）是指从一个节点运动到另一节点所耗费的行为总数。
- 分值（score）是指每一个已访问节点的代价与启发式预估值之和。这些已访问节点分布在到达当前节点的路径上。

现在让我们看看 A* 算法是如何应用这些术语的，或者简单地说，A* 算法是如何工作的。如你所知，A* 会找出两点之间的最短路径。但我们按照何种方式来测量呢？时间？距离？还是步数？尽管 A* 可以按照几乎任意一种测量方式来展开搜索，但我们选择使用距离。其他测量方式可能是时间（来找到花费走路时间最短的路径）或者耗油量（来找到消耗汽车油量最小的路径）。对于在一个区块与其水平或垂直方向的相邻区块之间运动的代价，我们赋予其值为 1（究竟是 1 英尺、1 英里还是一个区块的距离则是无所谓的）。从一个区块运动到其对角线上相邻区块的代价值为 1.41。1.41 是在一个对角线上相邻的两个区块的中心点之间的距离，也就是 $\sqrt{2}$ 。

h 值是从当前区块（你在搜索中正检测的那一区块）运动到目标区块所需代价的最佳猜测值。你可以用任何假设与逻辑来推测它，但当 A* 算法用于寻找最短路径时，*h* 值通常指的就是从当前区块到目标区块之间的直线距离。更确切地说，它是从目标区块中心到当前区块中心距离的最佳猜测值。你可以使用一些简单逻辑轻松地推导出该值。每个节点在被访问时都被赋予一个 *f* 分值：

$$f = g + h$$



注意 这里的直线式计算所得出的 *h* 值结果可能正确，也可能非常低（如果路径中有墙的话）。因而这种 *h* 值被认为是代价的低估值。低估值可以保证得到代价最低路径，但搜索可能会稍费时间。利用高估值来推导路径可能比低估值更快，但得到的也许不是代价最低路径。

让我们再说一遍，*h* 值就是当前区块和目标区块之间的距离的最佳猜测值。而这里的 *g* 值指的是到当前节点的路径上所有已访问节点的 *f* 分值总和。最好我们通过类比来理解一下。比方说你正计划一趟从纽约到巴黎的旅行。因为你的预算不多，所以你想找到一条花费最少的路线。你可以考虑将纽约、伦敦、里斯本、布鲁塞尔、马德里与巴黎作为节点。在你研究路线的过程中，你计算出从纽约到伦敦的花费并记下结果。你同样也计算出了从里斯本到纽约、从伦

敦到巴黎等的花费。最后，如果你使用了 A* 的其他法则（还没谈到），你就会找到最佳路线（从花费角度）。我们假设这条路线就是纽约—马德里—伦敦—巴黎。在纽约（其他节点也同样）时，f=g+h。记住 g 值是之前所有节点的 f 值之和。因为纽约是起始节点，没有更早的了，所以纽约的 g=0。接着你到了马德里，g 就不等于 0 了，因为它是由其他节点访问而来的。这时 g 值由来自纽约的 f 值组成。所以 g 值就等于到达当前节点的花费总额。如果你确实采用了这条旅游路线，那么 g 值就等于你走到当前位置所用的花费。

请记住 A* 不但能处理距离，而且还能处理地形。我之前提到过从一个区块到下一个区块的代价会是 1 或 1.14。但只有当所有区块都同样可成为最终路径的一部分时，这样计算才可行。假如一些区块由水形成，而除非迫不得已，你并不想让角色穿越水域。因此你会给包含水域的节点间转换代价赋予一个较大的值，比如是 10 吧。这不能保证路径不会从水域经过，但它会给予没有包含水域的路径一个极大的优先选择权。如果水域是流经整个地图的河流，而且那里没有桥，那么 A* 将最终会提供给你一条穿过水域的路径。但假如有座桥，而且它又比较近，那么 A* 会给你一条经过桥的路径。或者，假如角色是一个人鱼，那他可能更喜欢水路。在这种情况下，你可使到水域的代价比到陆地的代价更低。然而在绝大多数应用中，所有区块都具有同样的优先选择权，所以最低分值路径也就是最短距离路径。

10.3.2　伪码

前面，我们讨论了 A* 算法的基本概念以及它的工作原理。但我们还不了解该算法的详细处理过程，下面你会看到该算法的伪码。然后是针对每一行的解释。

```
1  AStar.Search
2       create open array
3       create closed array
4       s.g = 0
5       s.h = findHeuristic(s.x, s.y)
6       s.f = s.g + s.h
7       s.parent = null
8       push s into open array
9       set keepSearching to true
10      while keepSearching
11            pop node n from open
12            if n is the goal node
13                 build path from start to finish
14                 set keepSearching to false
15            for each neighbor m of n
16                 newg = n.g + cost(n, newx, newy)
17                 if m has not been visited
18                      m.g = n.f
19                      m.h = findHeuristic(newx, newy)
20                      m.f = m.g + m.h
21                      m.parent = n
22                      add it to the open array
23                      sort the open array
```

```
24                  else
25                      if newg < m.g
26                          m.parent = n
27                          m.g = newg
28                          m.f = m.g + m.h
29                          sort the open array
30              push n into the closed array
31              if search time > max time
32                  set keepSearching to false
33          return path
```

此算法用到了 open（开放）与 closed（封闭）这 2 种列表（在 Flash 中为数组）。开放数组包含从前已访问的节点。封闭数组包含所有已展开过的节点（也就是说，其所有邻近节点都已被访问）。我们将开放数组用作一个*优先级队列*——它不仅存储节点，而且还对其排序。我们将数组元素的排序方式定为从最低 f 分值到最高 f 分值。每当我们为开放数组新增一个节点或者改变其中一节点的 g 值时，我们都必须重新排序以保证节点的顺序规则不变。

在伪码的第 2 ～ 3 行中，我们创建了（空的）开放数组与封闭数组。接下来看一下在寻路中所需要的起始点与目标点。s 是代表起始节点的对象。我们设定 $s \cdot g$ 为 0，这是因为起始节点没有父级，所以得到的代价（g）为 0（第 4 行）。接着我们再找出起始节点的 h 值（请记住 h 值是从当前节点到目标点的代价估计值）。然后我们就可以存储起始节点的 f 值了，它是 $s \cdot g$ 与 $s \cdot h$ 的和（第 6 行）。由于 s 没有父级，所以我们设定 `s.parent` 为 `null`。最后我们把节点 s 添加到开放数组中（第 8 行）。于是节点 s 就成为开放数组中第一个也是唯一一个节点。

第 9 行我们设定 `keepSearching` 变量的值为 `true`。当其保持为 `true` 时，我们会一直运行 A* 搜索。当已找到一条路径或找不到路径，再比如当搜索时间过长时，我们就会把它设为 `false`。

在第 11 行我们从开放数组中取出一个节点 n，然后检查其是否为目标节点。如果是，则说明抵达目的地，于是停止搜索，然后建立路径（第 12 ～ 14 行）；如果不是，我们就将其展开，这意味着我们会访问它的每一个邻近节点。在第 16 行我们推导出 n 相邻节点 m（也即我们当前检测节点）的 g 值。接着我们检查是否已访问过 m 节点。如果还没有，那就进入第 18 ～ 23 行部分。先设定 m 的 g 值——也就是其父节点 n 的 f 值，然后计算并存储 m 的 h 值与 f 值，最后还要再将 m 节点的 parent 属性设为之前的节点 n。如果 m 节点之前被访问过并且现在有一个更低的 g 值，那么我们进入算法第 25 ～ 31 行的部分。

在此我想多谈谈 g 值。当节点被首次访问时，其 g 值是基于到达该节点路径所推导出的（如前所述）。但是有可能（甚至很有可能）在搜索中我们会通过另一可行路径再次访问该节点。如果它在新路径上得到的 g 值小于之前存储路径上的 g 值，那我们就以新换旧（第 27 行）。另外，在第 26 行我们设定 m 的 parent 属性为我们前一步经过的节点。在搜索结束后我们使用 parent 属性来建立最终路径。我们可以从目标节点通过跟踪 parent 属性来回溯到起始节点。接下来我们重新计算 f 值，并对开放数组重新排序。这是因为我们刚刚用了一个更小的 f 值更新了

其中一个节点，而该节点的优先级要比另一节点的高。

在访问过 *n* 的所有邻近节点后，我们就进入第 30 行。在这行中，我们将 *n* 添加到封闭数组中，这是因为它已完全展开。然后我们检测搜索占用时间是否太长。如果我们已搜索很久还没有建立路径，则设定 `keepSearching` 为 `false`。否则，我们就继续检测开放数组中的下一个节点（第 11 行）。如果 `keepSearching` 为 `false`，我们就会停止搜索并建立路径。

恭喜！你已经正式跨入 A* 算法之门了！当然，如果你觉得难于理解，也不要难过。毕竟它并不简单。我也是花了一番工夫，在网上读了很多相关文章后才完全理解了 A* 算法的基础的。

10.3.3　寻路范例

现在该来看看用 A* 寻路的范例了。我创建了一个文件，它能使你自定义一个 20×15 像素大小的网格并在其上检验 A* 寻路的结果（图 10-6）。



注意　可在 book_files/chapter10/astar 目录下找到此范例文件。

在讨论代码与如何使用 A* 工具类之前，让我们先来看看范例文件的特性吧。

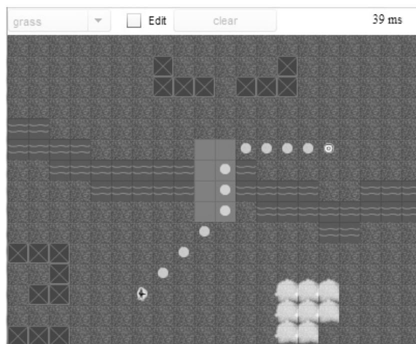


图　10-6

1. 特性

该范例有两种模式：编辑模式与测试模式。你可以通过点选 Edit 复选框在两者间切换。如果你想测试而非编辑，范例则会以预定义布局来进行初始化。

测试时，只需点击任何一个区块来设定起始点，然后点击其他任何一个区块来设置目标区块。程序会立即用 A* 搜索并寻找连接两个区块的代价最低路径。你还能在顶部的文本字段中看到寻路用时。大部分路径会需要耗费 10 ms ～ 100 ms 的时间。使用 A* 寻路虽不是很快，但还可以接受。因为在大部分应用中它只需每过几秒运行一次即可。

请注意，这个范例还包含了多种地形。在这里我们将其一一列出并将其可被用于路径的可取性等级予以标明。稍后我们还要研究下每一种地形的转换代价。

- ❒ 草地（grass）——最可取。
- ❒ 桥梁（bridge）——与草地一样，最可取。
- ❒ 水域（water）——可取性较低，如必要时还是可取的。
- ❒ 火（fire）——尽量避免选取，除非只有这条路。
- ❒ 墙壁（wall）——尽量避免选取，除非只有这条路。

A*算法的运作方式

请根据以下这 3 幅图（图 10-7 至图 10-9）来理解 A* 算法的运作方式。

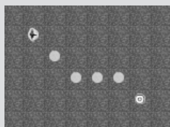


图 10-7 生成的一个只包含一种简单地形的典型路径

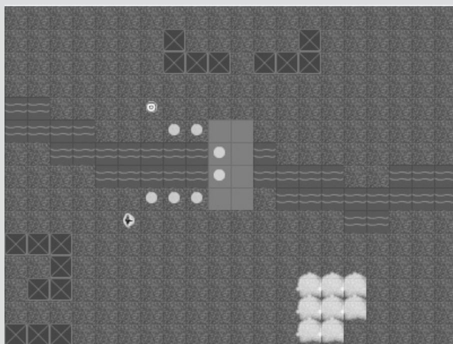


图 10-8 起始区块与目标区块间有水域穿过，但 A* 算法发现有桥并将其用在了最终路径中

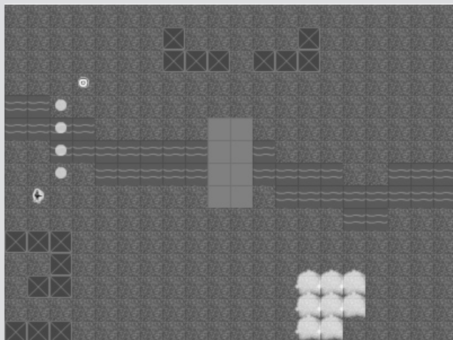


图 10-9 同上，起始区块与目标区块间有水域穿过，但如从桥经过，则路途遥远，所以 A* 算法确定了一条穿越水域的最低代价路径

如要编辑布局，请点击 Edit 复选框。通过点击 Clear 按钮你可以清除网格。要改变区块，可以从下拉菜单里选择一种地形，然后点击你想改变的区块。当如你所愿设置好网格后，取消选择 Edit 复选框并开始测试。

2. 代码

现在让我们来看看该范例要用到的一些代码。我们不会一一介绍构成 A* 算法的全部代码，而只介绍如何使用它。



注意 A* 工具类代码在 com.gamebook.utils.astar 类包中。

如果一个程序要使用 A* 工具类代码，它就必须用一个类来实现 ISearchable 接口，而且所有的区块也都必须实现 INode 接口。ISearchable 接口会确保将特定的方法与属性公开，以便于 Astar 类搜索实现了该接口的类。INode 接口能确保所有区块包含利于搜索的信息，比如地形类别或者在搜索过程中当前节点的 *h* 值。

当需要搜索时，我们先是创建一个 Astar 类的实例，接着将一个实现了 ISearchable 接口的类的引用设为参数，然后调用 search 方法，并传入起始节点与目标节点，最后该方法会返回一个 SearchResults 类的实例。

SearchResults 类包含一个用于表明搜索是否返回路径的属性。如果搜索确实返回了一条路径，那么 SearchResults 类的实例中同样也包含着这条路径，该路径是一个 Path 类的实例。

Path 类包含一个路径中所用到的全部区块的数组，当然也包括目标区块。它还包含该路径的总代价，这个也许你不会用到。

概括起来，你需要知道以下几点：你的应用程序要实现 ISearchable 和 INode 这两个接口，然后用 Astar 类来实施搜索，最后用返回的 SearchResults 实例来显示结果。

现在让我们来看一下怎样才算实现 INode 接口，接着再了解一下 ISearchable 接口。这个范例中的 Tile 类实现了 INode 接口。INode 接口要求 Tile 类中必须存在下列方法：

- getCol——返回区块的列序数；
- getRow——返回区块的行序数；
- getNodeType——返回区块的地形类型，比如草地或水域；
- setHeuristic/getHruristic——在搜索中设定和获取 *h* 值，你不需要做任何事；
- setNodeId/getNodeId——在搜索中设定与获取节点 ID，以便于更快地查找区块；
- setNeighbors/getNeighbors——在搜索中设定和获取，用以更快地找到当前检查区块的相邻区块。

Tile 类只要包含了以上所列的所有方法，也就能实现 INode 接口，从而在 A* 寻路中得以成功使用。

现在来看看 ISearchable 接口。这个范例中的 Grid 类实现了 Isearchable 接口，而实现该接口的类中要存在以下方法：

- getCols——返回网格总列数；
- getRows——返回网格总行数；
- getNode——返回行列序数已知位置的区块（实现了 INode 接口的）；
- getNodeTransitionCost——给定两个 Inode 类实例，返回转换代价（transition cost）。

当 Astar 类执行搜索时，它会用到上述方法。前三个非常好理解。需要注意的是 getNodeTransitionCost 方法。下面就是 Grid 类里的该方法：

```
public function getNodeTransitionCost(n1:INode,
n2:INode):Number {
  var cost:Number = costs[n1.getNodeType() + n2.getNodeType()];
  return cost;
}
```

该方法接受两个节点并返回它们之间的转换代价。例如，如果给定两个草地区块，该方法会返回一个很低的代价——1。注意到它是通过将两个地形类型串联成一个单独的键值（key）来寻找转换代价的，我们可在 costs 对象中检索到该键值。让我们来看一下完整的地形转换代价列表：

```
costs["grassgrass"] = 1;
costs["bridgebridge"] = 1;
costs["bridgegrass"] = 1;
costs["grassbridge"] = 1;
costs["grasswall"] = 1000000;
costs["wallgrass"] = 1000000;
costs["bridgewall"] = 1000000;
costs["wallbridge"] = 1000000;
costs["watergrass"] = 1;
costs["bridgewater"] = 10;
costs["waterbridge"] = 1000000;
costs["grasswater"] = 10;
costs["waterwater"] = 1;
costs["wallwall"] = 1000000;
costs["firefire"] = 1000000;
costs["firewater"] = 1;
costs["firewall"] = 1000000;
costs["firegrass"] = 1;
costs["firebridge"] = 1;
costs["waterfire"] = 1000000;
costs["wallfire"] = 1000000;
costs["grassfire"] = 1000000;
costs["bridgefire"] = 1000000;
```

在上面的这个长列表中，我们只用到了 3 个转换代价值。代价为 1 意味着转换是可取的，代价为 10 意味着转换虽不可取但如有必要还可使用，代价为 1 000 000 则意味着这种转换极不可取。

列表内含的基本逻辑可归结为几个简单规则：任何地形到草地的转换代价都是 1；草地到水域的转换代价为 10（不是很可取但如果需要也是可行的）；水域到水域的转换代价为 1，因为反正都已经湿了，角色可以保持湿漉状态直到走出水路，这也是可行的（低代价）。火与水不同——从烈焰之地到烈焰之地的转换代价仍然非常高。因为无论如何，你都不想变成“烧鹅”。不过话又说回来，这样确实能保证你在绝境中至少有条火路可走，而 A* 算法依然会将角色在火中穿行的时间减到最小（图 10-10）。



注意 列表中的这些转换代价值没有对与错的区分。这属于游戏设计的范畴——一切由你负责；你可以通过修改转换代价值来实现你所期望的游戏行为。

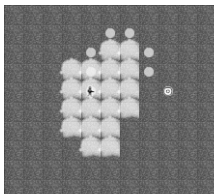


图 10-10　这种情况下的最佳路径（即代价最低路径，使得那个家伙最快地逃离火场的路径）并不是最短路径

最后来看看实际用来执行搜索的代码。在这个范例中，Grid 类也包含一个 UI 组件以及用来调用搜索与显示搜索结果的代码。当起始点和目标点被设置且执行搜索时，Grid 类里的 search 方法会被调用。以下是该方法中的一段代码：

```
var astar:Astar = new Astar(this);
var results:SearchResults = astar.search(INode(startTile),
➔ INode(goalTile));
if (results.getIsSuccess()) {
  var path:Path = results.getPath();
  for (var i:int=0;i<path.getNodes().length;++i) {
       var t:Tile = Tile(path.getNodes()[i]);
       t.showPath();
  }
}
```

第一行创建了一个 Astar 类的实例，传入参数 this——一个对 ISearchable 类的引用。下一行调用了 Astar 类实例里的 search 方法，并传递起始区块和目标区块给它。如果搜索成功地返回一个路径，那么接下来程序遍历路径节点并最终使得路径显示在屏幕上。

你可能困惑于为何 SearchResults 类会包含一个 success 属性——为什么它不直接返回路径呢？这是因为，运行一次搜索可能会占用很长时间。一般来说，它应该在 100 ms 以内。但如果网格比较大并且很复杂，那它可能就要占用几秒的时间。因为 Flash 并非多线程（意味着它在同一时间只可以处理一件事情），所以一个用时很多秒的搜索就会“冻结”程序直到完成搜索。Astar 类会让你给定一个超时值。默认值是 2 s。如果一个搜索占用时间超过 2 s，它将自动停止，并返回一个 suceess 属性为 false 的 SearchResults 类实例。



注意 由于游戏中会大量地用到 A* 算法，所以你能在网上与一些书中碰到很多与之相关的信息。通过学习这些知识，你就能让 A* 的处理速度变得更快，或者你会用更特别的方法来使用 A* 算法。我推荐你读 Ian Millington 写的 *Artificial Intelligence for Games* 一书。你可以在那本书里学到更多关于 A* 算法的知识。

第 11 章

合作游戏

并非所有多人游戏都会让玩家们相互竞赛。合作游戏（cooperative game）就是让玩家们组成团队来达成共同目标。开发合作游戏的挑战性是开发单人游戏或竞技型多人游戏时感受不到的。

除了少数例外（比如《战地双雄》（Army of Two）)，一般多人游戏不会按照纯粹的合作方式来进行游戏，而是在现有游戏设计中添加进合作的元素。例如，一款 FPS 类游戏可能会在标准的“死亡竞赛”（Deathmatch）之外再添加一个“夺旗”（Capture the Flag）模式。

我们要在本章讨论各种合作游戏的概念，并且会详细地讲解一个范例：“超级泡泡兄弟”（Super Blob Brothers）。玩家必须在该游戏中默契配合才能通关。

让我们先从合作游戏的几种类型开始讲起吧。

11.1　合作游戏的类型与方式

合作游戏可分为不同的类型，有时也可分为不同的游戏方式。

11.1.1　合作游戏的类型

合作游戏的类型是由玩家们所联合对抗的对象的类型而定。每种类型都包含着一套独特的游戏元素并且规定了该种游戏类型所需要的目标类别。没有哪个游戏能完全地符合这些类型中的任意一种。实际上，通常很多游戏都会混合使用这些游戏元素。让我们看看合作游戏的 3 种主要类别。

1. 玩家对抗玩家

在很多合作游戏中，玩家们能组队相互对抗，这其实是衍生自对战游戏的玩法。第一人称射击类游戏（FPS）就是“玩家对抗玩家”类型的范例。注意，就游戏机制而言，这种玩法与纯对战游戏的玩法稍有不同。合作关系是通过玩家们协同攻击战胜敌对方玩家而形成的。

“玩家对抗玩家”的游戏例子有《雷神之锤》（同一组内的玩家争取获得最大杀敌数），《命

令和征服》（任一团队对任意其他团队展开进攻与防御），《Wii 网球》（双打以获得最高分数）。

2. 玩家对抗 AI

在这个类型中，组队玩家要和受游戏逻辑控制的人工智能敌人进行较量。除了第一人称射击游戏之外，RPG 游戏与卷轴过关游戏也能被归入此类。玩家间必须相互合作，以此来进攻指定的目标或者对抗如潮水般多的敌人。你会发现很多游戏都有某种形式的“玩家对抗 AI”玩法——从《魂斗罗》到《暗黑破坏神》，玩家们组队对抗由游戏自身所生成的野怪。

3. 玩家对抗环境

“玩家对抗环境”这种玩法让团队解决来自于他们周边环境产生的挑战。玩家们必须解开谜题或者管理资源，而不是对付敌人。比如，玩家们可能必须同时站在两个不同区块上才能打开一扇紧锁的大门。《塞尔达传说——四剑风云》（The Legend of Zelda: Four Swords）就包含了这类合作元素，本章的游戏“超级泡泡兄弟”也是如此。

11.1.2　合作游戏的方式

有很多方法能使多人游戏采取合作式玩法。在本节中我们会尝试着把类似的游戏玩法归类。再次提醒，大部分游戏都不能被完全地归入任何一种类型中。

1. 能力对称类：多多益善

在采用这种方式的游戏中，玩家们具有完全相同的能力。玩家越多，则意味着力量越大。手持机关枪的两位玩家可以输出一位玩家双倍的伤害。

有了额外的队友，你们就可以轻松搞定非常强大的敌人，也可以轻而易举地战胜来自四面八方的大量敌人（图 11-1）。

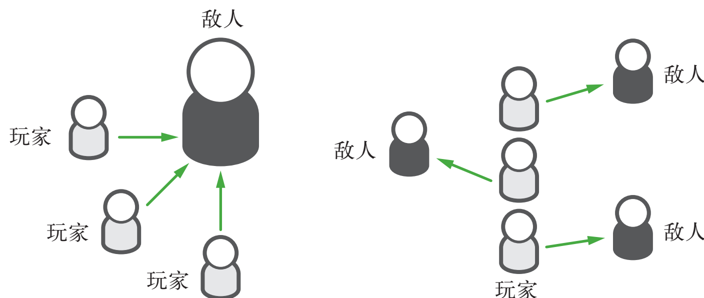


图　11-1

很多多人合作游戏都可归入此类，这其中包括像《光晕》与《雷神之锤》这样的 FPS 游戏、《合金弹头》与《魂斗罗》这样的横版射击类游戏，以及《1942》与《雷电》这样的纵版射

击类游戏。

2. 能力不对称类：休戚与共

在采用这种方式的合作游戏中，玩家们必须要使用各自独特的能力来互相支援。例如，一个玩家的进攻半径可能很短，但会对敌人造成重创，而另一玩家可以使用远距离武器，虽然对敌人造成的伤害较小，但贵在射程远。

通常采用这种玩法的游戏并不需要额外的队友支持就具有完整的可玩性。很难找到一个多人游戏真的要求玩家必须使用互补的能力才能完成游戏。

《圣铠传说》（Gauntlet）与《魔兽世界》中都存在一些能力不对称性。在这类游戏中，玩家必须选择加入一个阵营，所属阵营不同，玩家所能使用的技能也将不同。

3. 资源管理类：互通有无

在这种游戏玩法中，玩家可能拥有相同的技能，但会得到不同的资源。玩家们必须相互配合，通过交易各自所需资源来完成共同的任务。例如，玩家 1 得到了能发动玩家 2 的坦克的燃料。相反地，玩家 2 的位置接近于玩家 1 用来发电的宝贵矿藏。

4. 复杂机械类

这是最后的类型了。在这种类型的游戏中，我们需要额外的手（或机械爪之类的）来解决挑战。游戏环境的布局使单独一个玩家不能成功地完成操作。设想有一扇需要同时用两个门卡开启的大门，或者需要两个人的重量才能降下来的升降机（图 11-2）。复杂机械与"玩家对抗环境"类游戏是密切相关的。

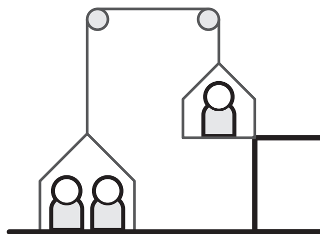


图 11-2

11.2 游戏："超级泡泡兄弟"

"超级泡泡兄弟"是一款"玩家对抗环境"类的双人游戏。它通过使用复杂机械类元素来强制两位玩家齐心协力共赴挑战。要知道玩家是很熟悉游戏的这些惯用手法的，比如像推岩石，在组队任务需要时扭转一个特殊的机关。旧有的游戏机制现在变得既新鲜又富有挑战性。



注意 这个游戏的源文件可以在 book_files/chapter11/blob_bros_game 里找到。

在"超级泡泡兄弟"（见图 11-3）中，玩家们必须穿越每一个关卡以抵达终点台（Goal Pad）。路上会有很多障碍物，包括闸门、岩石与激光塔（我们会在下面讲它们的属性）。为了进入下一关，两个玩家必须都站到终点台（Goal Pad）上。当通过所有关卡时游戏就胜利结束了。如果任何一个玩家生命值降为零，则游戏以失败告终。

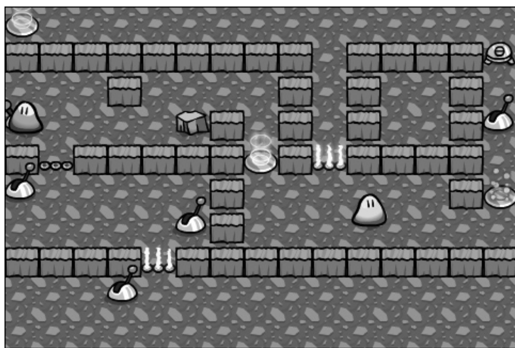


图　11-3

这个游戏需要玩家相互合作，因为大部分障碍都不能通过个人来克服。比如，玩家必须为对方打开闸门来打通通向终点之路。

游戏中有两种玩家：防守者和进攻者。服务器会自动为你选择玩家类型。你的玩家类型将总是和你队友相反。被激光塔瞄准时防守者会转变成一面盾牌，使得他无法被激光束穿透；而进攻者在激光塔的射程范围之内会转变成一把利剑，以摧毁激光塔。这样，两个玩家的能力互相弥补。他们需要协同作战以生存下去并完成游戏。

每个玩家有 3 条命。如果玩家在还有命的情况下死亡，它会在关卡的起始点或最近保存的重生点获得重生。

在这个游戏里，你用键盘上的方向键来控制两个泡泡兄弟的其中一个。游戏视角为顶视角且游戏是基于区块构建的。

下面就是你和你的搭档将遇到的障碍。

静止闸门和控制杆——静止闸门被激活时，玩家无法通过。静止闸门旁边有一个对应的可使闸门失效的控制杆（图 11-4）。把你的泡泡放到靠近控制杆的地方，以便拉动控制杆打开闸门。注意，控制杆很沉！从控制杆旁走过后它将会翻转回默认位置，从而重新激活闸门。

激光塔——当你不小心踩了发着红光的触发台时，激光塔将向你的泡泡发射激光束（图 11-5）。一旦被激光塔锁定，你的泡泡就无法躲避激光束。防守者踩到触发台不会受到伤害，但是攻击者会被杀死。

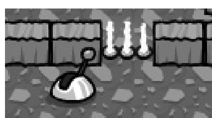


图　11-4

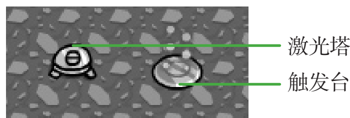


图　11-5

岩石——岩石经常阻塞泡泡的通道，或者位于去往杠杆或触发台必经之路上。幸而你的泡泡可以推走这些岩石，由于岩石很重，所以必须两个玩家一起（朝同一个方向）推来打开通道（见图 11-6）。每次推动岩石可以让其滑过一个区块。

存盘台——发着黄光的存盘点（图 11-7）可以使你保存状态。走到存盘台上将更新你的重生点，使得你不必原路返回。注意到达并激活存盘点只需要一个玩家。



图 11-6

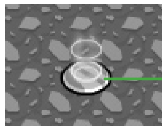


图 11-7

终点——发着绿光的终点台（图 11-8）代表每关的结束。我们全部的目标就是到达这个触发台，但这里有一个问题，即两个玩家必须同时站在这个终点台上才可以进入下一关。你要非常小心，以保证你和你的同伴都能到达终点台。

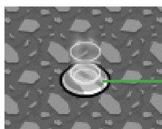


图 11-8

既然你熟悉了该游戏的机制，要注意将“超级泡泡兄弟”与你熟悉的游戏相比较。如果你玩过《罗罗历险记》（Adventures of LoLo）或《塞尔达传说》，那么你应该很熟悉这种玩法了。不过得再提醒你一下，当游戏变成多人游戏时，它会从整体上添加进新的特点。除了陷阱和谜题之外，与同伴的合作也会是个挑战。

11.3 服务器端与客户端：谁来决策游戏逻辑

在多人游戏中，开发者要设计好哪些游戏机制由服务器端管理，哪些功能可以交由客户端来控制。我们已经在第 6 章详细地探讨过这个问题。由于“超级泡泡兄弟”这个游戏完全是合作性而非竞争性的，所以它不必由服务器端来掌控一切细节。你可以将它与“玩家对抗玩家”类型的游戏相比，在那种游戏中，竞争可能会驱使玩家们采取一切可能手段来获得哪怕是一小块的边界。在那种情况下，让服务器端去验证运动是很重要的。而在我们这个游戏中，作弊不是什么大问题，所以我们只在必要时使用服务器端来进行验证。不同的游戏会要求不同的客户端与服务器端决策机制，而一个特定的游戏则需要你谨慎地在两者之间做出恰当的平衡。

如之前所说，我们来看下在“超级泡泡兄弟”里客户端与服务器端是如何协同工作的。

11.3.1 客户端

客户端会告知服务器端以下信息：

- 关卡初始化的详细信息；
- 我的角色的位置；
- 我的角色何时死亡；
- 我的角色何时到达一个存盘台；
- 我的角色何时到达终点台；
- 我的角色何时到达触发台；
- 我的角色何时尝试去切换控制杆；
- 我的角色何时尝试去推岩石；
- 我的角色何时摧毁一个激光塔。

11.3.2 服务器端

服务器端会告知客户端以下信息：

- 控制杆是否被成功地扳开；
- 静止闸门的状态；
- 岩石可否被推动；
- 激光塔何时开火；
- 关卡何时被完成；
- 每一玩家剩余的生命条数；
- 重生点是否已被更新；
- 何时及何处重生；
- 何时游戏结束。

11.3.3 理解游戏原理

本节我们将会仔细地讲解该游戏中的一些关键性原理，并且通过它们来更好地理解该游戏中的客户端与服务器端间的关系。

1. 切换开关：控制杆、触发台及闸门

切换开关是用来控制一些可以启动或关闭的游戏元素的。在“超级泡泡兄弟”中，控制杆、终点台与触发台都是切换开关。这些元素的每一个都可以自由开关，一个或更多的玩家可以通过启动切换开关而与之互动（图 11-9）。

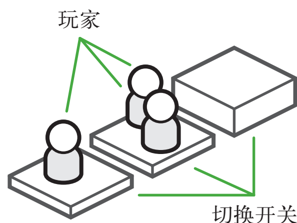


图　11-9

下面是控制杆与触发台如何作为一个切换开关来使用的。

- 控制杆：拉动 = 打开，没有拉动 = 关闭。
- 触发台：压下 = 打开，没有压下 = 关闭。

静止闸门与控制杆有紧密的联系。静止闸门可能没有或有几个控制杆。拉动一个或多个控制杆会使对应的静止闸门失效。如果松开一个控制杆，而且也没有其他的与此闸门有关联的控制杆被拉动，则该静止闸门将被重新激活。

由于控制杆的状态只可能是被拉动或没被拉动，所以服务器端需要及时了解控制杆状态并将其反馈到每一个玩家的屏幕上。如果玩家 1 拉动了控制杆，则玩家 2 只有等到当所有玩家松开控制杆后才可以拉动它。不过，当玩家 1 走开时，玩家 2 可以抓住控制杆来保持它的扳开状态。

服务器端需要确定是否有一个或多个控制杆被扳开。如果有，静止闸门保持失效状态。只有当所有的控制杆都被放开后，服务器端才会告诉客户端去重启静止闸门。



注意　记住，如前所述，控制杆是切换开关的一种。拉动一个控制杆就会使静止闸门失效。但如果拉动随后用于控制同一个门的另一个控制杆，前一控制杆虽会被成功扳动，但是门将保持原状。

2. 岩石

因为只有两个玩家完全一致地操作才能推动岩石，所以服务器端需要确定两个玩家是否朝同一个方向推动岩石。每个客户端都会告知服务器端其正在推动一块特定的岩石，以及推动的方向。如果两个客户端是朝同一方向推动，则服务器端会告诉客户端那个岩石已被挪开。如果任何一个客户端告诉服务器端他不再推了，那么这块岩石就无法被挪动。

3. 存盘台和重生点

客户端告诉服务器端何时抵达存盘台。服务器端会保存存盘台的位置。当玩家死亡后，服务器端会在经过一段规定的时间间隔之后通知玩家重生。服务器端会将存盘台位置以及重生信

息一起发送给客户端。

4. 终点台和关卡完成

当客户端抵达终点台时，它就会将该情况通知服务器端。既然要求两个玩家同时站在终点台上来通过一个关卡，那么服务器端就会留意有多少个玩家在终点台上。当终点台上的玩家数量达到 2 个，服务器端就会将一个“Level Complete”（关卡完成）的信息发送给每个客户端。

11.4　游戏消息

就像其他范例那样，该游戏的客户端和服务器端通过互相发送格式化的 EsObject 对象来进行通信。每个 EsObject 对象包含一个用于表示消息意图的 `ACTION` 变量，以及消息所需要的其他数据。表 11-1 包含了在“超级泡泡兄弟”中的每一种消息及其描述。

表 11-1 “超级泡泡兄弟”中的游戏消息及其描述

消　息	方　向	目的与行为
INIT_LEVEL	客户端向服务器端 / 服务器端向客户端	当客户端成功加入房间并准备去接收游戏数据时，它会发送布告板状态信息给服务器端。每个客户端都发送这个消息，但只有第一个会被使用 服务器端发送布告板状态信息给每个客户端以便于两个玩家都使用同样的数据
INIT_ME	客户端向服务器端	在客户端接收到布告板状态信息并使用那些数据初始化游戏后，客户端告诉服务器端它已经准备好可以开始了
PLAYER_LIST	服务器端向客户端	当一个客户端加入游戏，服务器端发送所有已经进入游戏的玩家的列表给客户端
ADD_PLAYER	服务器端向客户端	当一个新玩家加入游戏时发送给所有的客户端。也会在每关开始时发送给每个玩家
REMOVE_PLAYER	服务器端向客户端	当一个玩家离开游戏时发送给所有的客户端
POSITON_UPDATE	客户端向服务器端 / 服务器端向客户端	当玩家使用方向键运动到新位置时，会将该位置发送给服务器端。当玩家运动时，每半秒会更新一次坐标。服务器端保存新坐标并将此信息发送给客户端，以使同伴的位置与服务器端保持一致
FLIP_SWITCH	客户端向服务器端 / 服务器端向客户端	当一个玩家尝试切换开关时，该尝试会被发送给服务器端。服务器端根据开关的已知状态来验证该尝试。然后将翻转切换开关是否成功以及切换的是哪一个开关的信息发送给客户端
DESTROY_TOWER	客户端向服务器端 / 服务器端向客户端	客户端告诉服务器端要摧毁一座激光塔。服务器端确认数据并保存塔的最新状态。随后该激光塔的状态被回送给客户端以使图像能够反映出该塔已经被摧毁。因为这时服务器端已存储了该塔已被摧毁的情况，所以对于随后收到的客户端向该激光塔射击的消息将不予理会

（续）

消　息	方　向	目的与行为
PUSH_ROCK	客户端向服务器端 / 服务器端向客户端	将客户端去推岩石的尝试告知服务器端，以及在什么位置朝哪个方向上推。当其停止推动时，客户端会把同样的消息发送给服务器端 服务器端验证是否有两位玩家同时向同一个方向推岩石，并据此判断岩石是否被推动，以及被推到什么位置
PLAYER_DIED	客户端向服务器端 / 服务器端向客户端	当我的角色阵亡时告诉服务器端，服务器端告诉所有的客户端该信息，还有当前阵亡玩家的失去的生命条数
REVIVE_ME	服务器端向客户端	当玩家阵亡，如果还有生命条数，玩家经过短暂时间会复活。这个信息会告诉所有的客户端在布告板上的一个特别位置来重置该客户端
UPDATE_SPAWN_LOCATION	客户端向服务器端	当一个玩家走到存盘台上，这个信息会告知服务器端存盘台的位置。服务器端存储这个位置并将其作为任一玩家随后的重生点
LEVEL_COMPLETE	客户端向服务器端 / 服务器端向客户端	当一个玩家到达终点台，客户端会告知服务器端其已站到终点台上了。如果玩家走出终点台，该消息会重新发送给服务器端，提醒服务器端那个玩家已经不在终点台上了。只有当两个玩家同时站在终点台上时，服务器端才会反馈这条信息给所有的客户端，告知他们已通关
GAME_OVER	服务器端向客户端	当任一玩家的生命条数减少到零时，服务器端会告诉所有客户端游戏已结束以及谁是失败的根源
ERROR	服务器端向客户端	如果有错误发生将告诉客户端

11.5　客户端细节

如上节所述，在“超级泡泡兄弟”这个游戏中，客户端中存在很多行为，接下来我们来看看值得关注的一些细节。

11.5.1　初始化关卡

关卡是由客户端所加载的 XML 文件所定义的。描述每个关卡的 XML 文件所包含的信息有布告板尺寸、玩家的起始位置、游戏元素的位置以及切换开关与闸门之间的关系。

11.5.2　玩家的位置

“超级泡泡兄弟”使用第 7 章提到过的“我在这里”技术。我们之所以选择这种方法，一方面因为它简单易用，另一方面也是由于该游戏并不要求精准地同步。我们允许客户端在短时间内所显示的玩家位置有轻微的不同步，只要两个玩家赶得上彼此就行。

每个客户端会间隔半秒发送其所属玩家的位置，请看下面的代码段。你可以在 Game.as 中的 enterFrame 方法中找到该段。注意如果玩家没有运动，位置是不会发送的，因为与服务器端的这种通信是不必要的。



注意　Game.as 和 coopgame.as 不是同一个文件。

```
//send a position update every 500ms
if (getTimer() - _lastTimeSent > 500) {
        if (_myPlayer.x != _myPlayer.lastReportedX ||
        ➔ _myPlayer.y != _myPlayer.lastReportedY) {
        dispatchEvent(new PositionUpdateEvent
        ➔  (PositionUpdateEvent.POSITION_UPDATE, _myPlayer.x,
        ➔ _myPlayer.y));
        _myPlayer.lastReportedX = x;
        _myPlayer.lastReportedY = y;
  }
  _lastTimeSent = getTimer();
}
```

11.5.3　切换开关、静止闸门与激光塔

回想一下我们正在使用的切换控制杆、触发台与终点台状态的概念。当一个玩家走到包含一个切换开关的区块上时，该玩家会自动尝试去开启开关。一旦发生该事件，那么就有一条关于此次尝试的信息被发送给服务器端。

为了确定一区块是否包含开关，当游戏初始化时，我们在区块里保存开关的 ID，并命名为“trigger”（触发器）。如果一个区块没有包含开关，那么其 trigger 属性的默认值为 -1。无论何时，只要玩家一改变了区块，我们就会检查影响到的开关。

不仅仅是我们走到一个有开关的区块时会提醒服务器端，当离开那个区块的时候也会通知服务器端。随后服务器端就会将此开关恢复为默认状态。幸而我们可以使用相同的事件并发送相同的消息给服务器端。我们会传递一个布尔值，其值为 false 时关闭开关，为 true 时开启开关。以下是一段位于 Game.as 的 checkKeys 方法中的代码：

```
// handle any triggers
if (currentTile.trigger > -1) {
  dispatchEvent(new AttemptToggleSwitchEvent
  ➔ (AttemptToggleSwitchEvent.TOGGLE_SWITCH,
  ➔  currentTile.trigger, false));
}

if (attemptedTile.trigger > -1) {
  dispatchEvent(new AttemptToggleSwitchEvent
  ➔ (AttemptToggleSwitchEvent.TOGGLE_SWITCH,
  ➔  attemptedTile.trigger, true));
  _soundManager.playSound(SoundManager.STRAIN);
}
```

还要注意到当玩家尝试开启开关时，我们会播放一个音效。这个小技巧会带给玩家一种拉动控制杆的感觉。另外，你要记住，因为我们使用“我在这里”方法，所以有可能短时间内你同伴的位置会与你稍微不同步。为了避免任何意外，当一个切换开关用做一个静止闸门的控制杆时，服务器端会延迟一秒发送切换消息。这样使得消息返回到客户端的时候，所有玩家已经各就各位，如图 11-10 所示。

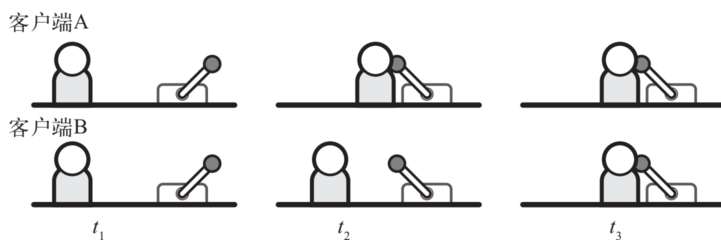


图 11-10

然后，我们发送一个切换开关状态消息给服务器端。通过该消息，我们也可以传递开关的ID，服务器端就能以此分辨出我们尝试切换的是哪个开关。要实现这些功能，我们得使用下列函数（在 CoopGame.as 中）：

```
private function onToggleSwitchAttempt(atse:AttemptToggleSwitch
➔ Event):void {
  trace("onToggleSwitch: " + atse.switchId + ", " + atse.isOn);

  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔ FLIP_SWITCH);
  esob.setInteger(PluginConstants.SWITCH_ID, atse.switchId);
  esob.setInteger(PluginConstants.SWITCH_STATE, atse.isOn ==
  true ? 1 : 0);
  sendToPlugin(esob);
}
```

接下来，对我们切换开关的尝试进行一些验证之后，服务器端会返回一个关于开关现状的消息。如果有任何的静止闸门或者激光塔与该开关有关联，消息里也会包含关于相关障碍物件的信息。将此信息返回给客户端的函数位于 CoopGame.as 中，如下所示：

```
private function handleFlipSwitch(esob:EsObject):void {
  // the switch that was flipped
  var switchId:int = esob.getInteger(PluginConstants.
  ➔ SWITCH_ID);
  var switchState:int = esob.getInteger(PluginConstants.
  ➔ SWITCH_STATE);
  _game.toggleSwitch(switchId, switchState);

  // who fliped it?
  var playerNameWhoFlippedSwitch:String = esob.getString
  ➔ (PluginConstants.NAME);
```

```
    // the results of what was flipped
    var switchResults:EsObject = esob.getEsObject
    ➔ (PluginConstants.SWITCH_RESULTS);

    // was it a gate or laser tower?
    if (switchResults.doesPropertyExist(PluginConstants.GATE_ID)) {

         var gateId:int = switchResults.getInteger
         ➔ (PluginConstants.GATE_ID);
         var gateState:int = switchResults.getInteger
         ➔ (PluginConstants.GATE_STATE);
         _game.toggleGate(gateId, gateState);

    } else if (switchResults.doesPropertyExist(PluginConstants.
    ➔TOWER_ID)) {

         var towerId:int = switchResults.getInteger
         ➔ (PluginConstants.TOWER_ID);
         var towerState:int = switchResults.getInteger
         ➔ (PluginConstants.TOWER_STATE);
         _game.setTowerState(towerId, towerState,
         ➔ playerNameWhoFlippedSwitch);
    }
  }
```

11.5.4　推岩石

推岩石是“超级泡泡兄弟”里最吸引人的一个游戏机制。我们前面说过，因为岩石太重，所以运动它们需要两个玩家的通力合作。由此，每个客户端尝试推岩石时都必须告知服务器端，而服务器端需要验证这个意图以确保两个客户端在朝同一方向推动。

一个测试的窍门

在一台电脑上测试推动岩石机制是很需要技巧的。这里除了用代码来提示测试中的变化之外，还有个窍门，它可以让你用 Flash 播放器使测试变得更容易。点击单独一个玩家的 Flash 播放器窗口之外的地方会让该窗口非前端显示。如果你按住你的右方向键并向右走，那么在非前端显示的窗口中你的角色还会继续向右走。所以要想测试推岩石的话，你只需在两个独立的 Flash 播放器窗口中分别打开两个游戏实例，让其中的一个角色推岩石，然后点击窗口之外，然后再让另一个角色也开始推岩石就 OK 了。这还真得要谢谢 Flash！

用于放置岩石的逻辑

由于“超级泡泡兄弟”是区块式游戏，而且有岩石的区块是不允许走进的，所以我们只能在其四周的区块上面运动岩石。要想推动岩石，我们只需检测要进入的区块是否存在岩石。如

果有的话，我们就会告诉服务器端要推动这个特定的岩石，并会用到如下代码（在 Game.as 中的 checkKeys 方法里）：

```
if (attemptedTile.hasRock) {
  if (!_myPlayer.isPushingRock && !attemptedTile.currentRock.
  ➔ isSliding) {
      _myPlayer.setIsPushingRock(attemptedTile.currentRock.id);
      attemptPushRock(attemptedTile.currentRock, _myPlayer.
  ➔ currentDirection);
      _soundManager.playSound(SoundManager.STRAIN);
  }
}
```

让我们来看看 attemptPushRock 函数。我们不仅需要注意推动的方向，而且要确定岩石不会被推进一个无效的区块（即任何一个不允许走进或不存在的区块）。例如，含有一堵墙的区块是无效的，因为这种区块不允许走进。允许玩家推动岩石进入一个墙壁是讲不通的。这个函数也在 Game.as 里。

```
private function attemptPushRock(rock:Rock, direction:int):
➔ void {
  var xMove:int = 0;
  var yMove:int = 0;
  var dirName:String;

  switch (direction) {
       case Player.DIR_NORTH :
            yMove = -1;
            dirName = "north";
            break;
       case Player.DIR_SOUTH :
            yMove = 1;
            dirName = "south";
            break;
       case Player.DIR_EAST :
            xMove = 1;
            dirName = "east";
            break;
       case Player.DIR_WEST :
            xMove = -1;
            dirName = "west";
            break;
  }

  var destinationTile:Tile = _grid.getTile(rock.currentTile.
  ➔ column + xMove, rock.currentTile.row + yMove);

  // is the destination tile ok to push a rock into?
  if (!destinationTile || !destinationTile.isWalkable) {
       trace("can't push rock there!");
  } else {
       dispatchEvent(new AttemptPushRockEvent
       ➔ (AttemptPushRockEvent.PUSH, rock.id, true,
```

```
        ➔ destinationTile.column, destinationTile.row,
        ➔ dirName));
  }
}
```

接下来，我们会把尝试推动的岩石的 ID、推动方向以及我们期望推动到的区块位置的 x 与 y 坐标值传送给服务器端。这要用到下面这个 onPushRockAttempt 函数，它位于 CoopGame.as 里。

```
private function onPushRockAttempt(apre:AttemptPushRockEvent):
➔ void {
  trace("onPushRockAttempt");
  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔ PUSH_ROCK);
  esob.setInteger(PluginConstants.ROCK_ID, apre.rockId);

  // only needed if we are trying to push
  if (apre.isPusing) {
       esob.setInteger(PluginConstants.X, apre.x);
       esob.setInteger(PluginConstants.Y, apre.y);
       esob.setString(PluginConstants.DIRECTION, apre.
       ➔ direction);
  }

  sendToPlugin(esob);
}
```



注意　从技术上来说，岩石只是“跳”到了终点——不是很生动。为了提升视觉效果，我们可以让岩石的图像在一小段时间内从原始点滑到目标点。（参见 Rock.as 里的 move 函数。）

最终，服务器端验证此推动尝试并告诉客户端岩石是否能被挪走。该尝试只有在当服务器端意识到两个玩家在同一方向推动同一个岩石才是有效的。哇！搬动一块小小的石头得做这么多工作！最终推动岩石的函数同样在 CoopGame.as 里。

```
private function handlePushRock(esob:EsObject):void {
  trace("handlePushRock");
  var rockId:int = esob.getInteger(PluginConstants.ROCK_ID);

  var tileX:int = esob.getInteger(PluginConstants.X);
  var tileY:int = esob.getInteger(PluginConstants.Y);
  var dirName:String = esob.getString(PluginConstants.
  ➔ DIRECTION);

  _game.pushRock(rockId, tileX, tileY, dirName);
}
```

11.5.5　结论和扩展

本章我们讨论了不同类型的合作游戏以及一些合作方式，我们还创建了一个真正可运行的游戏。

“超级泡泡兄弟”可以作为“玩家对抗环境”类游戏的一个范例。我们可以将这其中的许多游戏机制作为基础并加以改进，从而创建出功能齐全的游戏。

下面是关于如何改进“超级泡泡兄弟”现有思路与技术实现的几点意见：

- 对玩家位置使用时间同步，让其更精确地同步；
- 保存关卡数据在服务器端里而不是从客户端中加载；
- 由服务器端来判断玩家是否死亡；
- 增加更多的能力，比如允许泡泡牵引对象。

第 12 章

等距视图技术

自 20 世纪 80 年代中期开始，等距视图技术（isometric view）就被广泛应用于电子游戏中。你也许知道它被叫做“2.5D”或者“3/4 视图”。无论怎么称呼这项技术，它其实只是一种特殊的 3D 视图，有助于游戏开发者在这种视图中布置并控制游戏中的对象，这种 3D 视图不会像其他 3D 视图那样需要很多的计算开销。无数的热门游戏都采用了这项技术，比如《暗黑破坏神 2》（Diablo II）与《皮克民》（Pikmin）。

我曾碰到过很多 Flash 虚拟世界都使用了等距视图技术，比如 VMTV（virtual.mtv.com）和 Sifaka World（sifakaworld.com）。也有一些虚拟世界例外地使用了侧视图技术，比如 Poptropica（poptropica.com）和迪士尼的 Pixie Hollow（pixiehollow.go.com）。但如果你打算用 Flash 打造一个虚拟世界，就极可能会用到等距视图技术。所以要专心点，因为你将在本章学到这项技术的基础知识。

下面将介绍等距视图技术及其背后的几何原理。我们将了解应用这项技术所带来的一些好处，并会了解如何在范例中使用它。随着探讨的深入，对于那些在 3D 坐标系和屏幕坐标系之间映射坐标的公式，我们就不再作过多的解释了。

到本章结束时，你应该能掌握如何使用所提供的 `Isometric` 类来协助布置区块网格，如何在屏幕坐标系与虚拟世界坐标系间映射坐标，以及如何正确地按照立体深度来排列对象。

12.1 等距视图技术的基础知识与优点

“摄影机”这个概念被现今绝大多数 3D 游戏所采用。摄影机能在 3D 环境中到处运动（与旋转）。摄像机看到什么，屏幕上就会显示什么。等距视图将摄影机视角固定为一组指定的角度（我们将在 12.2 节讨论这些角度值）。这组视角允许我们观察虚拟世界并放置素材元件，它不仅运算快捷而且也非常方便。

12.1.1 等距视图中的对象

图 12-1 所示为一个长宽高分别与 x、y、z 轴对齐、在等距视图中显示的正立方体。

你可以注意到，在这个视图中，立方体每边的长度相等。这也是“等距”这一名称的由来。现在来看看同样的立方体摆放在不同的几个位置会是什么样子（见图 12-2）。

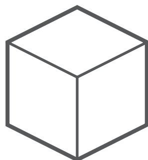


图　12-1

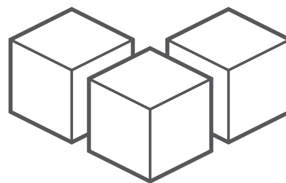


图　12-2

上面的图示说明了两个非常重要的等距视图概念。

❑ 透视不变——在等距视图中，当我们在场景不同位置摆放同一个立方体时，它总会显示相同的面。而在非等距视图中，随着立方体的运动，立方体的某条边看起来会比它左边或右边的邻边要长一些，如图 12-3 所示。就表现游戏内容而言，等距视图的对象显示一致性确实很棒：无论观察视角如何改变，我们只需要创建一次对象，然后就能在很多地方重用它们。

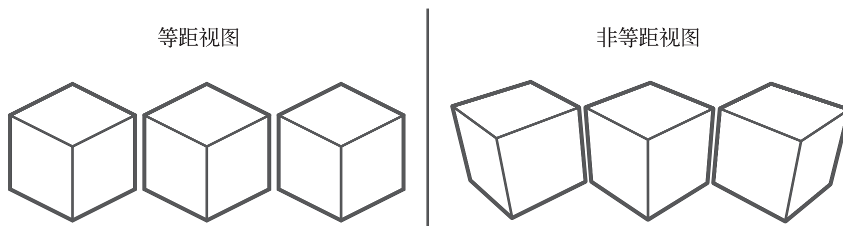


图　12-3

❑ 没有灭点——正如你在图 12-3 中看到的那样，在等距视图当中，没有视平线，也没有灭点（图 12-4）。这意味着当对象被放置在屏幕上时，它们的尺寸不需要缩放。

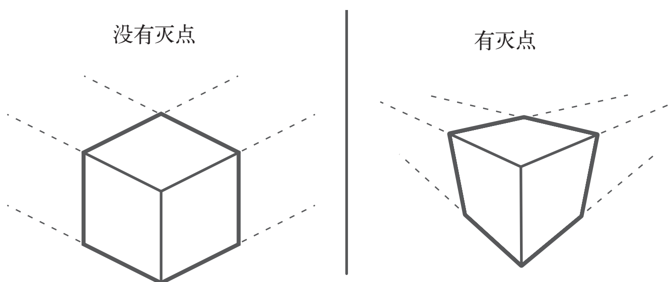


图　12-4

由于有了这两个主要优点，所以我们就能用位图（或矢量图）来充当要摆放在虚拟世界中的可视物件（见图 12-5），因为它们的尺寸不会缩放也不需要改变视角。而这将极大提升程序性能，因为可视物件不再需要直接通过实时渲染 3D 模型得出了。

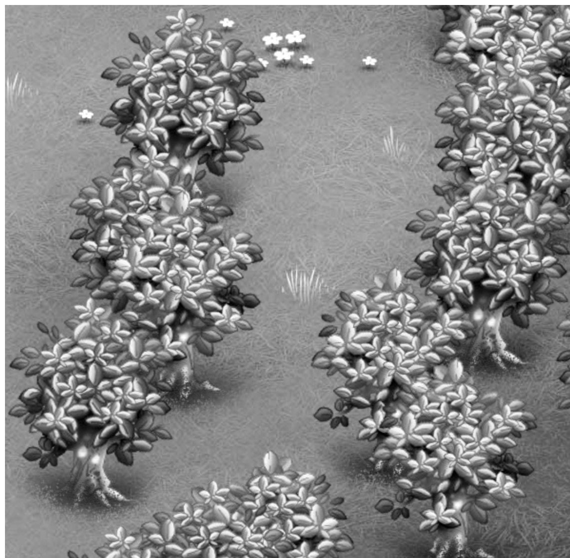


图　12-5

12.1.2　区块

由于等距视图不需缩放尺寸或改变透视就可重用同样的物件，我们可以通过铺贴区块来创建地图，这方法确实不错。图 12-6 所示地图就完全是由区块构成的。



图　12-6

在等距视图中，区块形状为菱形。你可以使用 Flash 或者其他绘图工具轻松地创建出这种菱形区块，按照以下 3 步（图 12-7）即可：

(1) 绘制一个任意大小的正方形；
(2) 把这个正方形旋转 45°；
(3) 把旋转后的正方形的高度缩放为原来的 50%。



图 12-7 这个菱形就是等距区块，它可以在创建网格时被任意重用

本章和本书后面章节还将屡次提到区块尺寸的另外几个基本制式。首先，菱形区块的宽度[①]是高度的两倍。其次，区块的尺寸比例应该是 2∶1。常见的区块尺寸是 64×32 像素与 128×64 像素。在 12.2 节中，我们将更多地讨论对区块尺寸选择以及错误选择所带来的影响。而现在你只需要知道一点：上面所列尺寸恰好能使区块在铺贴时很好地排列在屏幕上。

12.1.3 虚拟世界范例

本章关注的是可视化地使用区块与物件创建而成的等距视图环境。在第 10 章中我们提到过，混合使用区块式与绘制式技术来创建游戏与虚拟世界通常是种不错的选择。现在请看下面这两个虚拟世界，我们将介绍其每一个所用到的视觉方案。

1. Precious Grirls Club

这个虚拟世界完全由区块创建而成——其中一些区域多达 90 000 个区块！当屏幕卷动及化身运动时，区块可被直观地添加进屏幕或被移除（图 12-8）。



图 12-8

① 本书中菱形区块的宽度与高度和菱形的数学表述不同，在本书中区块宽度指的是菱形中最长对角线长度。另一个对角线的长度即为高度。——译者注

2. Faraway Friends

这个虚拟世界混合使用了区块式与绘制式这两项技术，在内存中使用区块以方便数据存储，但背景用的是自行绘制的图片，如此庞大且绚丽的背景要比屏幕可见区域大很多（见图 12-9）。这个虚拟世界中也有可重用的对象，不过场景没有用区块搭建。



图　12-9

12.1.4　等距视图技术的更多话题

等距视图技术中值得研究的问题要比本书涉及的内容多得多。本书内容足够你应对绝大多数虚拟世界的需求，但我们不打算涉及下列几项技术。

- 上方 / 下方——化身能够走过一座桥，而且随后它能走到这座桥的下面。这种技术在等距视图中实现起来要比听上去难得多。不过它挺有趣，值得解决，所以你可以试试看。
- 斜面——想想《疯狂弹子球》（Marble Madness）[①] 吧。具有斜坡感的区块能够使平坦的世界变得更加丰富（有深度感和材质感，同时也更加的复杂）。
- 分层排序——想象一下，毯子铺在地板上，而椅子压住了部分地毯，化身能够走过地毯或者坐在椅子上。一个功能完备的虚拟环境是很值得采取这样复杂的排序方式的。

① 一款由 Atari 公司于 1984 年发布的街机游戏，程序部分是由 Mark Cerny 与 Bob Flanagan 完成的。玩家使用轨迹球控制屏幕上的弹子球在一定时限内越过障碍物与敌人抵达终点。除此之外，它还是游戏史上第一款真正采用立体声音效的游戏。——译者注

12.2 技术视角

迄今为止，我们已了解了使用等距视图技术的几个好处，但我们没有介绍它的坐标系以及如何使用 ActionScript 来实现这些概念。是时候稍微正式地讲一下等距视图技术了。我们将在本节中介绍 Flash 坐标系统与等距坐标系，同时还会学习如何使用 `Isometric` 类。

看完这一节后，你将能够使用 ActionScript 创建一个由区块组成的网格。

12.2.1 几何原理

本书至此，我们已经知道坐标系有两个维度：*x* 轴与 *y* 轴。Flash 是没有 *z* 轴的（如果它真有 *z* 轴的话，那么 *z* 轴正方向将会电脑屏幕的背面延伸出去）。

我们可以概念化地将等距视图理解为（并随后以数学方式来处理）一种位于 Flash 坐标系下的 3D 次级坐标系。这里我们把次级坐标系称为“等距系统”，而把首要坐标系称为“Flash 系统”。Flash 系统是固定不变的，也就是说不能移动，因为它是限定在你的电脑显示器屏幕范围内的。当以某个方位把等距系统安置到 Flash 系统当中之后，等距系统就会一直保持等距的特性。请注意，当方位发生变化的时候，等距系统是不会发生改变的。唯一要弄明白的是怎么在 Flash 坐标系统中创造等距的效果。

现在，让我们假设等距系统跟 FLash 系统完全对齐。在这种情况下，这两个系统是没有任何差别的。事实上，现在还不能称它为等距系统。那怎样才能让第二个系统在 Flash 系统中看起来等距呢？通过执行以下两个步骤即可解决。

(1) 先围绕其 *x* 轴旋转 30°。将 *x* 轴作为旋转轴，这样当坐标系旋转时它会保持不变。在旋转前，等距系统的 3 个坐标轴与 Flash 系统的 3 个坐标轴完全对齐。旋转后，等距系统的 *x* 轴依然与 Flash 系统的 *x* 轴保持一致，而其他两个坐标轴则不再对齐。

(2) 再围绕其 *y* 轴旋转 45°。将 *y* 轴作为旋转轴，这样当坐标系旋转时它会保持不变。旋转完成后，等距系统中的 *x* 与 *z* 轴与其起始位置不同，最终我们就得到了一个从 Flash 系统中看到的等距系统。



注意 book_files/chapter12/rotation_animation/animation.swf 就是这个旋转动画文件。

你可以参看我们提供的 animation.swf 文件。这个动画文件可使你形象地理解这两次旋转是如何进行的。开始时它会显示出 Flash 系统的正向坐标轴方向，然后分两步旋转一个立方体。当动画完成时，等距视图中的立方体就会显示出来。

图 12-10 显示了从 Flash 坐标系统看到的最终的等距坐标系坐标轴方向。

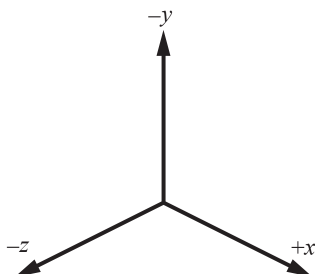


图 12-10　在此空间中，你可将由 $-z$ 与 $+x$ 组成的象限形象化地理解为地面。如果某物被拉离地面向上升起，那么它是沿 $-y$ 轴方向运动的

12.2.2　Isometric类

我们刚刚看到了显示在 Flash 坐标系中的等距坐标系。当要编写等距视图游戏时，你会发现我们经常需要把坐标从等距系统（现在我们称其为 3D 坐标系）映射到 Flash 系统（从现在我们只称其为屏幕坐标系），或者将坐标从屏幕坐标系映射回 3D 坐标系。而这就需要用到 `Isometric` 类。



注意　你可以在 com.gamebook.utils 包中找到 `Isometric` 类。

`Isometric` 类能够轻松地将给定的屏幕坐标转化为相应的 3D 坐标，反之亦然。除了对一些变量计算一次（留待以后使用）以外，该类不存储任何信息。它就像一个黑匣子，接受一个输入，然后返回一个输出。下面来看看该类的构造函数及其两个方法。

下面即是构造函数：

```
public function Isometric() {
  var theta:Number = 30;
  var alpha:Number = 45;
  theta *= Math.PI/180;
  alpha *= Math.PI/180;
  _sinTheta = Math.sin(theta);
  _cosTheta = Math.cos(theta);
  _sinAlpha = Math.sin(alpha);
  _cosAlpha = Math.cos(alpha);
}
```

如果你还记得上节内容，那你就该明白等距坐标系是由 Flash 坐标系旋转两次得到的，先旋转 30°，再旋转 45°。我们把这两个角度分别称为 `theta` 和 `alpha`。接着再将其值从角度制转换为弧度制，这样就能够在三角函数中使用它们。后面 4 行代码计算并存储它们的正余弦值，以备将来使用。

现在让我们来看看 `mapToScreen` 方法：

```
public function mapToScreen(xpp:Number, ypp:Number,
➔ zpp:Number):Coordinate {
  var yp:Number = ypp;
  var xp:Number = xpp*_cosAlpha+zpp*_sinAlpha;
  var zp:Number = zpp*_cosAlpha-xpp*_sinAlpha;
  var x:Number = xp;
  var y:Number = yp*_cosTheta-zp*_sinTheta;
  return new Coordinate(x, y, 0);
}
```

该方法接受一组3D坐标值，随之将其映射到屏幕坐标系并返回相应的屏幕坐标值。在数学上，当涉及多重坐标系时，我们通常会在每个新坐标系的 *x*、*y*、*z* 轴符号前标注 ′ 号（原文为prime，意为上标），这样就能确定变量所属的坐标系。所以你也就明白上列代码中 `p` 与 `pp` 所代表的含义了，它们分别是“prime”与“prime prime”的首字母缩写。上述方法中的变量 *x* 与 *y* 属于屏幕坐标系，变量 `xpp`、`ypp` 与 `zpp` 属于3D坐标系。而变量 `xp`、`yp` 与 `zp` 则是到一个新的临时坐标系的中介映射坐标。换句话说，我们在对先前所做的30°与45°变换实行逆旋转。（这种变换的数学原理暂且不论。）注意到返回结果是一个 `Coordinate` 类的新实例。此类只是用来储存 *x*、*y*、*z* 坐标值的。另外，你肯定会发现 *z* 坐标值为0。这是因为屏幕坐标系只是一个二维坐标系，故 *z* 值必须为0。

下面是 `Isometric` 类的第二个也是最后一个方法。

```
public function mapToIsoWorld(screenX:Number,
➔ screenY:Number):Coordinate {
  var z:Number = (screenX/_cosAlpha-screenY/
➔ (_sinAlpha*_sinTheta))*(1/(_cosAlpha/_sinAlpha+
➔ _sinAlpha/_cosAlpha));
  var x:Number = (1/_cosAlpha)*(screenX-z*_sinAlpha);
  return new Coordinate(x, 0, z);
}
```

该方法取得屏幕坐标，将其映射到3D等距空间，然后返回一个包含映射结果的 `Coordinate` 类实例。这个方法经常用来将鼠标点击位置的坐标映射到虚拟世界中，以确认点击了何种对象。它也常用于鼠标拖放行为。接着我们把一组屏幕坐标作为参数传入该方法中，其后两行代码就能将其映射到3D空间中。这两行看上去可真复杂！最后返回包含映射结果的 `Coordinate` 类实例。你会发现其中的 *y* 变量被标为0。这是由于我们进行的是从2D到3D的映射，所以不可能凭空生成一个新的自由度。因此假如有一个2D坐标点要映射为3D坐标点，因为其原本就是处于前面我们所讲的“地面”之上，所以转换之后其 *y* 值也应该同样为0。

通过从数学角度来分析等距坐标系可知，为了定位物件并能到处移动，你所需要的就只是 `Isometric` 类中的这两个方法。为了对物件进行排序，你还需要写其他一些代码，本章后面将会予以讨论。

12.2.3　创建一个网格

让我们来看个范例，它将教会你如何使用 Isometric 类与正确排列的区块来创建一个 10×10 的网格。



注意　你可以在 book_files\chapter12\grid 目录中找到创建网格的相关文件。

在开始介绍代码之前，我想先讲一些问题，虽然它们不是很突出，但理解它们却很重要。当编写一个等距视图游戏时（就此而言，也可以是任何 3D 游戏），你不必采用任何特殊方法去记录数据。数据是保存在内存中的，其中的数学处理方式与数据如何被渲染并在屏幕上显现的方式毫不相关。只有当物件必须被添加到屏幕上或者在屏幕上更新时，我们才会用到等距概念。例如，一个小球或许位于 3D 空间中的 (10, 20, 30) 点处。你通过惯用的标准物理学代码所产生的重力或其他力能够不断地影响小球的位置。只有当需要在屏幕上更新小球位置时，你才需要考虑如何显示小球，而此时你应该获取小球位置的坐标值，然后将其代入 Isometric 类中的 mapToScreen 方法。

言归正传，继续我们的代码研究。我们主要关注 Map 类，它能用 Isometric 类来创建一个包含 100 个区块的网格（图 12-11）。

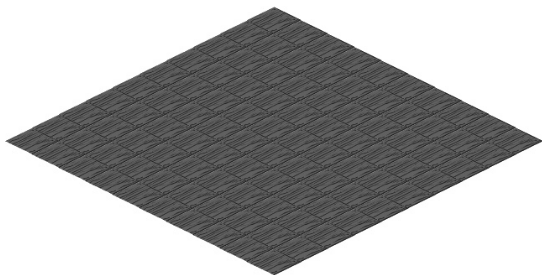


图　12-11

Map 类构造函数调用了 initialize 函数。它能完成所有使用 Isometric 类的程序所必须处理的工作。下面就是 initialize 函数的主要代码：

```
private function initialize():void{
  _iso = new Isometric();

  // 屏幕坐标系下的区块尺寸
  _tileWidthOnScreen = 64;
  _tileHeightOnScreen = 32;

  // 3D 坐标系下的区块尺寸
  _tileWidth = _iso.mapToIsoWorld(64, 0).x;
  _tileHeight = _tileWidth;
```

```
        // 建立区块网格
        buildGrid();
    }
```

第一行代码创建了一个 Isometric 类的实例，并将其存储到一个类属性中。接着，我们用两个变量存储了在屏幕坐标系中显示的菱形区块的宽度与高度，其值分别为 64 和 32（马上我们就会讨论这一点）。紧接着后面的两行代码计算并存储了它们在 3D 坐标系中的尺寸。通过使用 mapToIsoWorld 方法把屏幕坐标系中的宽映射到 3D 坐标系中，因为 3D 空间中的区块其实是个正方形，所以设置 _tileHeight 等于 _tileWidth。

确定3D坐标系中区块的正确尺寸

上面的代码看起来可能有点怪。我们先把区块设置为希望在屏幕上呈现的尺寸，然后再把它映射到 3D 坐标系中，这样我们就得到了能够映射为正确的屏幕尺寸的 3D 坐标系下的区块尺寸。这看起来似乎有悖“常理”。似乎我们应该先在 3D 坐标系下设置一些尺寸，然后将它们映射到屏幕坐标系下，而不管这样得到的屏幕尺寸是不是我们想要的。（特别是在前几段我还一直在讲数据驱动视图，而不是由视图来驱动数据，怕是更会加深了你对这种“常理”的认同。）

那么我们为什么要这样反其道而行之呢？还是先来看看如果不这么做会带来什么后果吧。在你转变认知之前，似乎正确的方法就是在 3D 坐标系下设置好尺寸，比如说 40×40 像素。接着你利用这些尺寸来计算出要显示到屏幕上的区块尺寸。一个 3D 坐标系下 40×40 像素的区块映射到屏幕坐标系就变成了 56.568×28.284 像素的区块了。随后你就可以让美工来尽可能地接近这个尺寸来绘制区块。

当区块网格建立好后，其中的区块都是基于 56.568×28.284 像素来进行摆放的。但是你会发现，区块的行与列不管向何方延伸，它们之间总会出现一条微小的裂缝，本来这些行与列应该是紧密地靠在一起的。这些裂缝在有些位置是逐渐增大的，而在另一些位置则是逐渐缩小的。这种差异是由两方面原因造成的，一方面这跟绘制的图片在屏幕上的显示方式有关，另一方面则跟图片坐标位置如何舍入最近的百分位有关。

你可能会遇到的另一个问题：如果你不是将区块作为单独的显示对象摆放到屏幕上，而是直接将它们绘制到一个位图当中，那么这些区块的位置坐标就被舍入到整数位。依据你摆放区块的方式（使用绝对的数学方法还是相对于前一个区块来进行摆放）而定，你很有可能会导致裂缝持续增大，而不是像前面我们说过的那种添加区块时出现的忽大忽小的裂缝。

这个问题困扰了我很多年，不过最终我还是找到了问题的症结所在以及解决办法。在非 Flash 平台下进行开发时也会出现这个问题。而解决办法就是**优先考虑屏幕尺寸**。你所选取的区块尺寸要恰好与像素完美匹配。除此之外，如果你选择的尺寸是 2 的幂的话，那么数学运算就会变快。这就是为什么我们所看到的区块一般都是 64×32 或 128×64 的原因。

好了，让我们回到代码，initialize 函数最后调用了 buildGrid 函数：

```
private function buildGrid():void {
  _grid = [];

  // 创建网格尺寸
  var cols:int = 10;
  var rows:int = 10;

  // 创建网格
  for (var i:int = 0; i < cols;++i) {
       _grid[i] = [];
       for (var j:int = 0; j < rows;++j) {
            // 创建区块
            var t:Tile = new Tile();

            // 区块在 3D 坐标系下的位置
            var tx:Number = i * _tileWidth;
            var tz:Number = -j * _tileHeight;

            // 将区块的 3D 坐标映射为屏幕坐标
            var coord:Coordinate = _iso.mapToScreen(tx, 0, tz);

            // 区块在屏幕坐标系下的位置
            t.x = coord.x;
            t.y = coord.y;

            // 储存区块
            _grid[i][j] = t;

            // 将区块添加到屏幕上
            addChild(t);
       }
  }
}
```

这个函数创建了所有的区块并将其摆放到屏幕上。它首先创建了一个名为 _grid 的数组以存储创建的区块。以后这些区块可以随时被引用。接着我们把要创建的网格尺寸定为 10×10 区块大小。列序号向右逐次增加，行序号向左逐次增加，如图 12-12 所示。

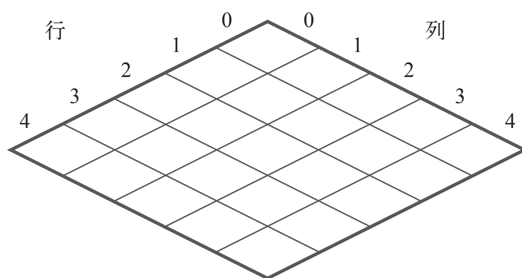


图　12-12

接着我们用到了一个嵌套循环。外层循环遍历所有的列，并在 _grid 数组中为每一列都创建一个新数组。内层循环则遍历当前列所对应的每一个行中的元素。

在内层循环中，我们首先创建一个 Tile 实例。它是显示对象。我们再根据当前的循环索引值以及 _tileWidth 与 _tileHeight 属性值计算出该区块在 3D 坐标系中的位置。请注意，此时区块应该位于由 $+x$ 与 $-z$ 组成的象限中。如果你还记得的话，该象限就是地面，而地面的 y 坐标值应该为 0。

我们现在得到了区块在 3D 坐标系下的位置，然后就可以利用 Isometric 类实例的 mapToScreen 方法将其坐标映射到屏幕坐标系中，根据所得的屏幕坐标，我们就能在屏幕上定位该区块。根据区块所在的行列序号，我们将区块按次序存储到 _grid 数组中，最终区块被添加到屏幕上并显示出来。

不要太精确

有时过于精确也会有问题。如果所绘制的区块像素尺寸与 64×32 分毫不差，那么在所有区块之间就会产生一条很细的线（见图 12-13）。

解决方法就是在所绘制的区块的宽度与高度上各加一个像素，但在排列区块时依然按照 64×32 来进行排列。将区块尺寸变成 65×33 就能消除那条小细线（见图 12-14）。

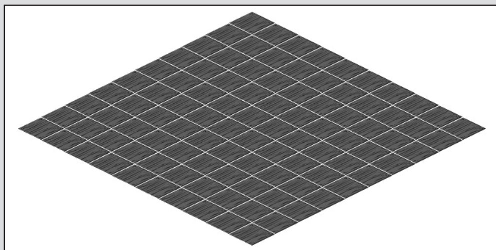


图 12-13

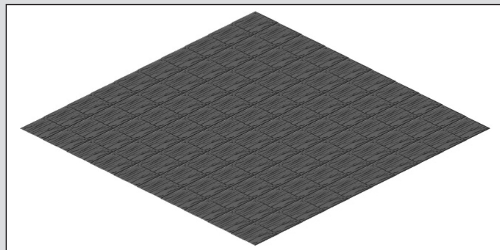


图 12-14

12.2.4 选择区块

现在我们通过一个范例来演示如何用鼠标输入来精确定位鼠标所经过的区块，这需要将鼠标屏幕坐标映射为 3D 坐标。当鼠标运动时，鼠标指针经过的区块就会被选中（见图 12-15）。

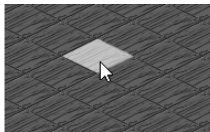


图 12-15



注意 你可以在 book_files\chapter12\select_tile 目录中找到相关文件。

该范例建立在前面范例的基础之上，因此大部分的代码我们已讨论过了。这里只看新增的内容，下面这行被添加到 initialize 函数中。

```
addEventListener(MouseEvent.MOUSE_MOVE, mouseMoved);
```

当鼠标在网格上运动的时候，mouseMoved 函数就会被调用。

```
private function mouseMoved(e:MouseEvent):void {
  if (_lastTile != null) {
      _lastTile.alpha = 1;
      _lastTile = null;
  }

  var coord:Coordinate = _iso.mapToIsoWorld(mouseX, mouseY);
  var col:int = Math.floor(coord.x / _tileWidth);
  var row:int = Math.floor(Math.abs(coord.z / _tileHeight));

  if (col < _cols && row < _rows) {
      var tile:Tile = getTile(col, row);
      tile.alpha = .5;
      _lastTile = tile;
  }
}
```

被选中的区块将会存储到类属性 _lastTile 中。该函数先把最后一次选中的区块的 alpha 属性设置为 1，然后将 _lastTile 设置为 null。这就从根本上保证了已选中的区块不会被再次选中。接着我们利用 mapToIsoWorld 方法将鼠标位置映射到 3D 坐标系中。然后根据鼠标 3D 坐标 x 与 z 值就能找到该点所在的行列序号。在第 10 章中我们介绍过一种简单的数学方法，它根据 x 与 y 坐标值立刻就能算出鼠标所在的行列序号，现在我们用的就是这种方法。但要注意的是，我们在确定行序号时使用的是 z 坐标的绝对值，这是因为我们虽位于由 $+x$ 与 $-z$ 所组成的象限中，但行与列的序号值一定为正值。

如果 col 和 row 的值在网格范围内，我们就能通过行与列序号值准确地找到该区块，然后将该区块的 alpha 值改为 0.5。最后，将该区块的引用存储到 _lastTile 变量中。

这个简单的例子足以说明如何把用户输入的屏幕坐标映射到 3D 坐标系中了。

12.3　排序算法

既然你已经弄明白了等距视图技术，并且知道如何去创建一个区块地图，理所当然，下一步就是往地图上填充物件了。将物件正确地放置到虚拟世界中并不难，但如何在等距视图中合理地对这些物件排序却长期困扰着众多开发者。术语“排序”（sort）在这里所指的是基于物件的 3D 坐标来对它们进行分层排列（图 12-16）。

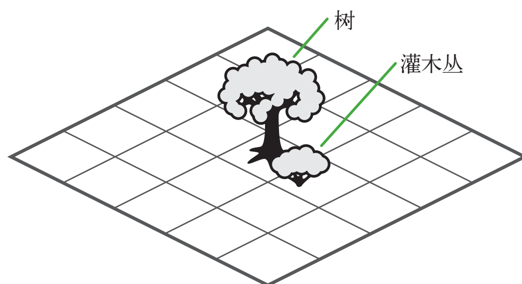


图 12-16　在这个图例中，灌木丛应该显示在树的前方，而树应该位于灌木丛的后面（依树的坐标而定）

尽管通过 Google 在网上到处搜索，但我仍然找不出一个令人满意的方案来处理等距视图中对物件排序的问题。不过我却在很多论坛中搜索到了大量的关于此问题的回复。本节中我将给大家介绍一种解决方法，然后通过一个范例来予以应用。

12.3.1　逻辑

我们想通过排序算法来实现的功能有：能够知道所有物件的坐标位置以及它们所占据的行列序号，并且会按照现实生活中的样子将它们正确地排序。

我们在此做两个假设。

(1) 所有物件都是由充满区块的矩形构成的。它可能只有单独一个区块那么大（1×1 个区块），或者会像餐桌那么大（可能是 2×4 个区块）。除此之外，我们不允许出现其他形状的物件，比如说 L 型的长沙发。如要使用这样的物件，则应该将它拆分为两个物件。

(2) 物件之间不能相互重叠。就跟现实生活中一样，你不可能让长沙发与餐桌占据同一块地方，这里我们也不允许出现这种情况。

现在介绍排序算法的工作原理。开始时我们会有一个包含所有待排序物件的列表（以下简称“物件表”）。然后我们再创建一个新的用以存储已排序物件的空列表（以下简称“排序表”）。接着我们就会遍历物件表并且将其中物件与排序表中的物件一一比较。如果在某次比较中，你发现物件表中当前物件的顺序应位于排序表中正查看物件之下或之后时，那么就将此物件添加到排序表中，就位于刚刚与其比较的排序表中的物件之前。比较不断地进行下去，直到物件表的所有物件都被添加到排序表后才结束。如果一物件没有排在任何物件之后，那就将它添加到排序表的末尾。最终我们就得到了一个序号由低到高的排序正确的列表。

这里要重点考虑的是，如何对两个物件进行比较以确定一物件是否应排在另一个之后呢？这是整个排序算法的核心。让我们来看看图 12-17 和图 12-18 吧。

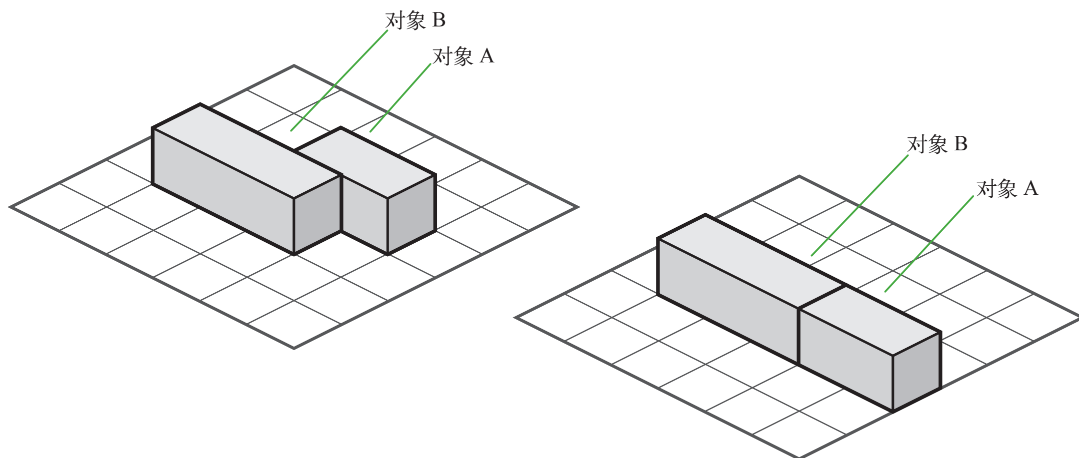


图　12-17

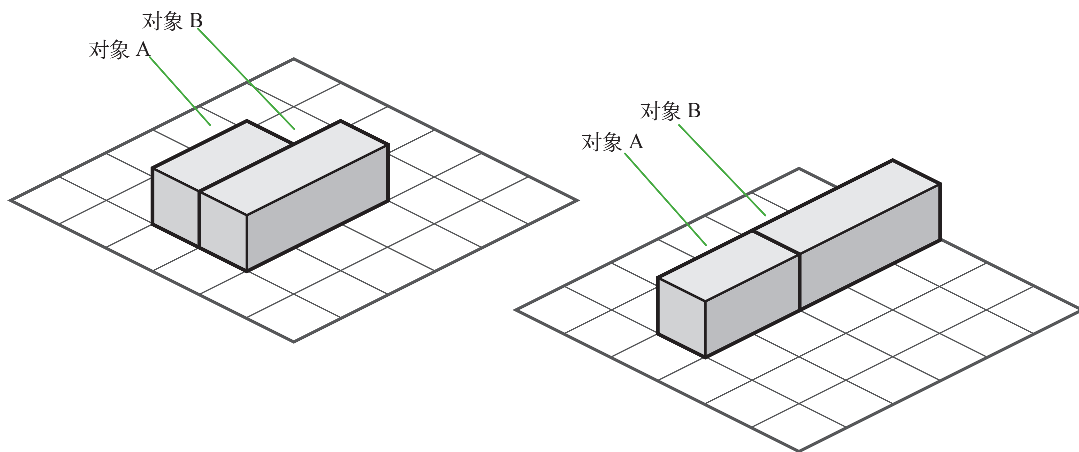


图　12-18

当通过比较来查看是否 B 物件应位于 A 物件之后时，我们可以先回答以下两个问题。

(1) B 物件的起始列序号是否小于或等于 A 物件所能扩展到的最大列序号？

(2) B 物件的起始行序号是否小于或等于 A 物件所能扩展到的最大行序号？

如果这两个问题答案都是“是”，那么 B 物件就应排在 A 物件之后。你可以花些时间对照上面这些图来自己想一下这两个问题并验证一下这个逻辑的正确性。

12.3.2　排序范例

在该范例中，我们将把物件添加到屏幕上然后对其进行正确的排序。下面范例中的物件看

上去放置得很正确（图 12-19），但我们没有对其进行排序。

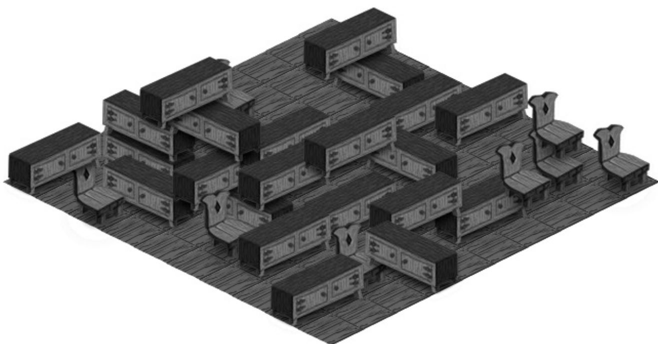


图 12-19



注意 你可以在 book_files/chapter12/sorting 目录下找到此范例源文件。

图 12-20 才是排序过的。

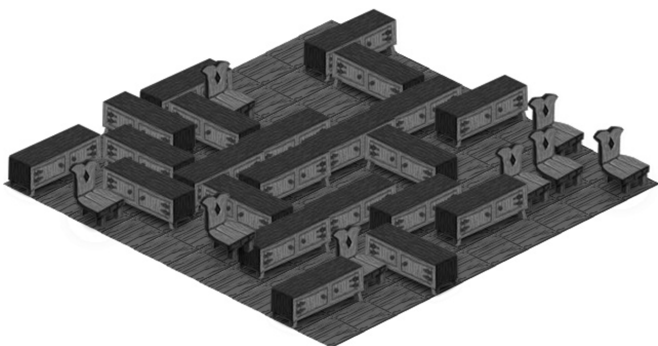


图 12-20

该范例是由本章早先的网格范例扩展而来的，因此我们只关注相应的新增代码。待添加到屏幕上的所有物件都被放进一个名叫 _itemHolder 的影片剪辑中。在 initialize 函数创建了网格之后，紧接着就添加了这个影片剪辑：

```
_itemHolder = new MovieClip();
addChild(_itemHolder);
```

并且这次我们在 initialize 函数的末尾又添加了如下代码：

```
for (var i:int = 0; i < 50;++i) {
  var col:int = Math.floor(_cols * Math.random());
  var row:int = Math.floor(_rows * Math.random());
```

```
    var item:Item = new Item();
    item.type = Math.floor(3*Math.random());

    if (testItemPlacement(item, col, row)) {
         addItem(item, col, row);
    }
  }

  sortAllItems();
```

通过 50 次迭代，上述代码试图将物件随机放置到不同位置。它会随机选取一行与一列，然后创建一个随机类型的 Item 新实例。Item 实例的类型可以是 0、1 或 2。Item 类中的 type 属性决定了物件会显示哪张图片，以及物件会占用多少行与列的区块。接着，我们用一个条件语句测试当前物件放置的位置是否正确。如该位置超出地图范围或其上已存在物件，那么该位置就不正确。如该位置正确，则我们会使用 addItem 函数将物件添加到屏幕上，然后调用 sortAllItems 函数对其进行排序。

下面是 addItem 函数：

```
private function addItem(itm:Item, col:int, row:int):void {
  for (var i:int = col; i < col + itm.cols;++i) {
       for (var j:int = row; j < row +itm.rows;++j) {
            var tile:Tile = getTile(i, j);
            if (tile != null) {
                 tile.addItem(itm);
            }
       }
  }

  var tx:Number = _tileWidth * col + _tileWidth / 2;
  var tz:Number = -(_tileHeight * row + _tileHeight / 2);

  var coord:Coordinate = _iso.mapToScreen(tx, 0, tz);
  itm.x = coord.x;
  itm.y = coord.y;

  itm.col = col;
  itm.row = row;

  _itemHolder.addChild(itm);

  _sortedItems.push(itm);
}
```

这个函数首先遍历可以容纳物件的区块，然后将物件添加进去。testItemPlacement 函数（一个用于测试物件摆放是否有效的函数）能够检查区块以确定其是否已经容纳了某个物件，这就是我们为什么能在这里将物件添加进区块的原因所在。接下来，通过使用与计算区块的 3D 坐标相同的技术，我们得出物件在 3D 坐标系中的位置，唯一不同的是，区块是沿着两个方向被添加到目标位置上的。算出物件的 3D 坐标后，物件就会占据在区块的中央。接着使用

mapToScreen 方法就能得出物件在屏幕上的位置。然后告知物件其所占据的行与列的序号，紧接着我们就把该物件添加到 _itemholder 影片剪辑中以便将其显示出来。最后，我们将物件的引用存储在 _sortItems 数组中（尽管现在物件还没有被排序）。

在继续往下讲之前，你应该知道所有的 Item 实例都实现了 ISortableItem 接口。这很重要，因为当你需要添加那些非装饰性物体对象（比如化身）到屏幕上时，你仍然需要下面这些相同的排序逻辑。化身也需要实现这个接口，这样它就能够像其他任何对象一样被排序。一个类只需实现下列方法即可算实现了 IsorttableItem 接口。

```
function get col():int;
function get row():int;
function get cols():int;
function get rows():int;
```

让我们一起来看看 sortAllItems 函数。

```
private function sortAllItems():void{
  var list:Array = _sortedItems.slice(0);

  _sortedItems = [];

  for (var i:int = 0; i < list.length;++i) {
       var nsi:ISortableItem = list[i];

       var added:Boolean = false;
       for (var j:int = 0; j < _sortedItems.length;++j ) {
            var si:ISortableItem = _sortedItems[j];

            if (nsi.col <= si.col+si.cols-1 && nsi.row
            ➔ <= si.row+si.rows-1) {
                 _sortedItems.splice(j, 0, nsi);
                 added = true;
                 break;
            }
       }
       if (!added) {
            _sortedItems.push(nsi);
       }
  }

  for (i = 0; i < _sortedItems.length;++i) {
       var disp:DisplayObject = _sortedItems[i] as
       ➔ DisplayObject;
       _itemHolder.addChildAt(disp, i);
  }
}
```

在这个函数中，我们继续实施上文提到的排序算法。首先，通过复制 _sortedItems 数组创建一个名为 list 的数组。接着，将 _sortedItems 数组重新设置成空数组。然后我们遍历 list 中的每一个物件，并用一个名为 nsi（new sortable item，新的可排序物件）的变量予以

引用。对于每一个物件，遍历 _sortedItems 数组中的所有物件，并对该数组中的每一个物件用一个叫做 si（sortable Item，可排序物件）的变量进行引用。对于每一个 nsi 对象，先设置一个相关的 added 变量并将其设为 false，以保证这个 nsi 以后能够被添加到 sortedItems 数组的末尾（如果在排序过程中该物件不会排到任何物件之后的话）。紧接着的条件判断语句包含了两个比较表达式，用来确认 nsi 是否应该排到 si 的后面。如果这两个比较表达式都为 true，则 nsi 被插入到 _sortableItems 数组并位于 si 之前，并将 added 变量设为 true，然后跳出内层循环。

该函数最后要做的就是遍历新得到的物件已排序的 _sortedItems 数组，并根据正确的立体深度把每个物件添加到屏幕上。最终，屏幕上的物件都按照正确的顺序被排列好了。

第 13 章

化　　身

从现在开始我们将专注于虚拟世界的研究，后续几章也同样如此。我们专为本书开发了一个虚拟世界——古老家园（Old World）。我们将用这个简单的模型来阐述有关虚拟世界的诸多概念。

我们把在虚拟世界中用来代表你的虚拟角色叫做“化身”（avatar）。你可以定制你的化身，并且控制它在虚拟世界中自由行走。化身是所有虚拟世界中的最核心的元素。

在本章中我们将讨论化身的典型组成部分以及几种不同的化身绘制方法，并利用一个较深入的范例来对其中一种绘制方法加以特别说明。最后，我们会用一个简单的程序来演示如何创建一个定制的化身，然后以这个化身的身份登入虚拟世界。这个程序是更大的“古老家园”项目的一部分。

13.1　了解化身

虚拟世界是社交网络，因此，玩家间能够互动并建立关系。他们可以通过很多方法会面和联络，而方式的多少只取决于虚拟世界设计师们的创造力。常见的互动方式有聊天、做游戏以及交易物品等。



注意　关于 Old World 核心的讨论将集中在本书后面这 4 章中。你可以在 book_files/old_world 找到相关的源文件。

当玩家们进行互动时，他们彼此间主要能接触到的就是对方的化身。因为它对于开启玩家间互动来说非常重要，并且又是你的虚拟形象代表，所以化身也成为了展现玩家个性的一个焦点。所有社交网络（虚拟的或者非虚拟的）的核心特点之一就是自我表现。玩家必须能让自己在某些方面独树一帜。如果达不到这一目的，那么他们很可能就会去玩另一个虚拟世界。

玩家主要是通过定制化身来展现其个性的（图 13-1）。虚拟世界本身所提供的功能决定了化身可被定制的程度。下面的范例列举了一些可定制的范围。

- 服装——化身定制中的一种常见类型。大多数虚拟世界都允许定制服装。玩家可通过服

装选项来定制上衣、裤子、鞋和帽子等。

- 肤色——如果化身是一种类人角色，那么通常它的肤色都是可选的。
- 头发——选取一个独特的发型可以为化身的整体形象增色不少。
- 体型——由于创建的内容牵涉很多方面，所以 Flash 虚拟世界很少提供体型的定制。但有时候还是可以选择的，比如可选择高矮和胖瘦。



图 13-1　Sifaka World（www.sifakaworld.com）里的一个全面定制的化身

另一种常见的自我表现方式是通过表情。表情就是化身用以表达情绪的一段特殊动画，比如说抛一个飞吻或者捧腹大笑。

通常化身都会有个*储物栏*——储存所有获得物品。这些物品可以是任何东西，它们或用于装扮自身形象（如上衣或者裤子），或用于娱乐（比如几首歌）。由于定制化身形象对于玩家想体现自我个性至关重要，所以通过获取物品满足定制与个性化需要就成为了玩家不断追求的目标。

获取物品也可以让虚拟世界的运营者从中赚钱。玩家可以用虚拟货币去购买物品。虚拟货币可以用真实货币来兑换（比如用信用卡买），或者通过花费很多时间在虚拟世界重复地做任务（比如玩游戏）来获取。

13.2　绘制化身的方法

将所需可定制部分装配成一个化身可算是一项挑战。我们先来了解几种绘制化身的方法，它们各有优缺点。首先，了解下列注意事项有助于你选择绘制方法。

- 业务需求——用来满足游戏设计的资源或功能，比如说化身的换装或者拿东西的功能。
- 制作流程——比如说，如果打算为化身的装备制作海量的素材，那么为了将来能降低处理素材所带来的计算消耗，我们就要在开始时选择最合适的开发方式来提高效率。
- 技术可用性——有时候，开发者会考虑用较先进的技术来装配并绘制化身，但 Flash 的速度或许还不能很好地应用该项技术（尤其对于我们将讨论的 3D 绘制方法来说）。

没有哪一种方法能完美地适用于所有虚拟世界。我曾经参与过 9 款虚拟世界的制作，并且见过或使用过几乎每一种装配与绘制化身的方法。我会将这些方法分作几个主要类别，并在下面对它们予以讲解。然后我们会单列一节来重点详述其中的一个方法——精灵序列图（sprite

sheet），并在 13.4 节中举例说明其用法。

13.2.1 木偶法

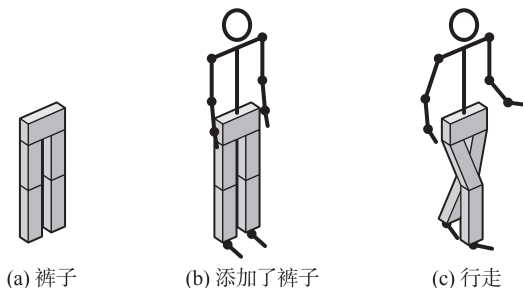
这种方法不太容易用语言表述清楚，有处理复杂 Flash 影片剪辑经验的开发者应该非常熟悉这种方法。

木偶法是用若干影片剪辑来组成化身并制作预先的动画，然后加载静态图片素材到这些影片剪辑中（或其上）的方法。化身的服装被分割成若干块，这些块与未穿衣服的动画躯体一一对应。比如图 13-2 中的这样一个化身的身体。

这个化身的裤子由 5 部分组成：胯部为一部分、左右大腿各一部分、左右小腿各一部分（图 13-3a）。这些静态图片会被加载进来并放到合适的影片剪辑中（图 13-3b）。利用这些影片剪辑制作化身行走的循环动画，同时图片素材也是跟着一起动的（图 13-3c）。



图 13-2



(a) 裤子　(b) 添加了裤子　(c) 行走

图 13-3

现在你就能将同样的方法推广开来了，你可以用它来制作诸如衬衣、帽子、头发以及任何其他你能想到的附属物。

1. 优点

- 文件体积很小——动画只需制作一次，外衣总是以静态图片的形式被加载到合适的位置上。也就是说，我们不用再对服装素材制作动画。因此，木偶法是所有绘制方法中文件体积最小的一种。
- 创建素材——由于素材只需是静态图片而无需再做动画，则对于内行而言，此法极为省时。（基于同样缘由，这也是个缺点，参见下文。）

2. 缺点

- 素材制作过程——美工需要很长时间才能掌握这种方法。这不是一种常规的方法。我们在 Electrotank 的几个项目中使用了木偶法，却发现该方法对于新美工而言是很难快速掌握的。

- 木偶制作过程——开始时要创建木偶的影片剪辑结构，这是一个漫长的过程。如果你想做好，有可能得花几个星期的时间。通常化身的结构会在 8 个方向上一一进行绘制并制作动画。再将结构分解为多个影片剪辑并对其一一命名以便程序调用。然后创建一些素材范例来检查所有影片剪辑的位置是否合适。
- 表现力不足——现在正开发的大部分虚拟世界对化身的外观有更高的需求，但这超出了木偶法所能达到的水平。化身看起来越逼真，它的可运动部分就越多，配置木偶结构所用的时间就会越长（更别提手工调节出在 8 个方向上模拟人类行走动画所耗费的时间了）。

3. 适用情况

企鹅俱乐部（www.clubpenguin.com）中的化身就很适合用此法制作。它们形象简单，需要运动的部分很少。这样化身定制也变得简单，并且文件体积也非常小。

13.2.2　叠层动画法

这种方式最容易理解。想象你有一个正走着的裸体化身。再拿一个已预先做好动画的衬衫，将其叠放于正走着的化身上面。结果你看到的就是一个穿着衬衫在走的化身（图 13-4）。

如图 13-5 所示，将一件裙子、一双鞋和一套头发与裸体化身组合起来，就得到一个打扮齐整的化身了。（耳环是头发图片的一部分。）

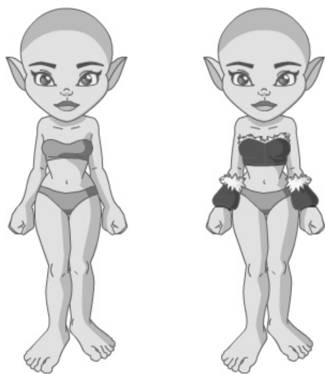


图　13-4



图　13-5

如果化身有一个基本的身体，然后将头发、外衣和鞋都叠放在上面，那么你可能会想该如何组织这些文件。通常有如下两种方法。

- 一件物品的所有信息都被放在同一个文件里。例如，所有的动画和它们的旋转效果都在一个 SWF 文件中，这个 SWF 会被加载并在需要时被播放。

❒ 关于一个物品的单独一种动画的全部信息被放在一个文件里。那么，行走动画对应的发型在一个 SWF 文件里，而跳跃动画对应的发型在另一个 SWF 文件中。

1. 优点

❒ 如果美工想手绘化身（不想用 3D 软件），那么用这种方法就能满足他的愿望，让你发挥出自己的特长。
❒ 学会创建内容不需要花费多长时间。绝大多数动画制作人员都能轻松地掌握这项技术。
❒ 编程实现很容易。

2. 缺点

❒ 仅限于手绘创作。如果你需要更逼真的化身，那么这种方法就不适用了。
❒ 既然是手绘，那么素材就是矢量图形。在 Flash 里矢量图形动画运行会比较慢。想要在屏幕内同时显示 20 个以上全部由矢量图形构成的化身就会使画面很卡。

3. 适用情况

如果你打算走个性的卡通风格路线，或者你碰巧有一位顶尖动画制作高手，那么这种方法就很适合你的项目。

13.2.3　精灵序列图技术

精灵序列图就是一张包含着大量图形信息的巨大位图。当然，我承认这样的解释太含糊。当我们将其应用于虚拟世界的化身上时，这个定义就会变得清楚起来了。现在你只需知道精灵序列图就是一个包含动画中所有单帧的网格就行了（图 13-6）。事实上，在所有平台上开发的多种游戏都曾用过精灵序列图技术。

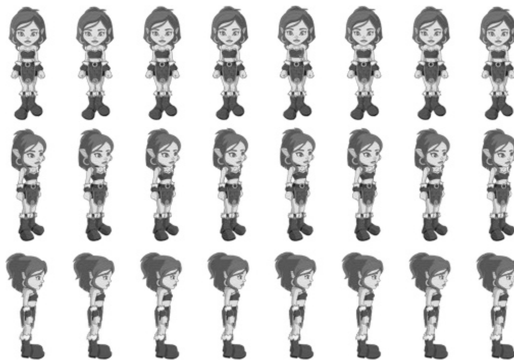


图　13-6



注意　在 13.3 节我们将深入讨论精灵序列图，本书中的范例也将用该方法来绘制化身。

类似于叠层动画方式，这里我们也是通过叠加多个精灵序列图来实现化身的定制。这一点我们会在 13.3 节深入讲解。现在只需了解所有的图像信息都一个个按网格依次排列在一张大图片文件里，借此 Flash 可以创建一个全身打扮好的并且带有动画的化身。

1. 优点

- 可实现虚拟世界中性能表现最好的化身。
- 可以很好地利用 3D 动画软件输出精灵序列图或者能处理为精灵序列图的格式。
- 方便编程。

2. 缺点

- 占用内存相当大。在 13.3 节中我们将看到这方面的计算过程。
- 与 UGC（User Generated Content，用户生成的内容）不能良好地协作，这一点将在 13.2.4 节中阐述。

3. 适用情况

在现有大多数 Flash 虚拟世界所采用的化身绘制法中，该方法可能是最好的。其性能优势是其主要优点。

13.2.4　3D渲染法

这是一种极具潜力的化身绘制法。使用该方法时，用户要先载入 3D 模型，接着赋予其材质，然后在客户端将化身渲染出来。还可以用 3D 骨骼动画来赋予模型动作。

近来 Papervision 3D 和 Away 3D 在 3D 渲染速度上取得了不小的进步，但遗憾的是，它们的性能仍然达不到在 Flash 中渲染所有化身的程度。以目前速度，我觉得用它们在定制界面渲染一个质量适中的单个化身是可行的。但想要渲染整个场景，我不认为这些 3D 引擎可以保证既能快速响应用户操作又可以同时渲染 20 个以上的化身。

1. 优点

- 多个 3D 模型可以共用一套骨骼动画。
- 可动态改变模型材质。
- 可以从任意角度渲染化身，用户定制时可以更自由地察看效果。
- 文件体积较小。
- 适用于 UGC 模式。用户可以自己为衬衣创建一个新图案，以此作为模型的动态材质。

2. 缺点

- 作为渲染整个虚拟世界的方案还不可行，性能达不到要求。

3. 适用情况

现在最好将这种方法只用于化身定制界面。虚拟世界中的化身则必须用另外的方法来绘制。如果将来 Flash 的 3D 引擎性能足够好的话，那么我强烈建议你使用 3D 渲染法来渲染化身。

13.2.5 试试视频

如你所见，我们有那么多方法可用于绘制化身，每一种都各有其优缺点。也就是说，没有一种是完美的。不过有一种方法我已经想了很久——视频。我们知道，3D 引擎的效率还不够高，而精灵序列图技术看起来是绝大多数项目的最佳方案，那么我们该如何改进精灵序列图呢？ Flash 视频（FLV）有可能会帮我们一把。精灵序列图包含着动画的所有帧，帧间变化极小。而视频则正是一种只记录帧间变化的视觉技术，因此用视频来承载精灵序列图的所有数据可能会使文件体积更小。

我还没能真正地使用过视频，但我确实对它还存在一些疑问：

- 有什么较好的方法能用来制作带透明通道的 FLV 文件？
- 在所含内容相当的情况下，PNG 格式的精灵序列图文件与 FLV 文件比起来哪种体积更小？
- 超过 20 个的视频实例同时播放对 Flash 的效率影响有多大？

由于它未经测试，所以也无所谓优缺点。但试一试应该会很有意思。

13.3 精灵序列图

精灵序列图（用于化身时）是一种巨大的网格式位图，其中的每一格都对应着一个动画截屏（图 13-7）。每一动画截屏都是化身动画的一帧。精灵序列图通常采用 PNG 格式，这样就可使用其中的透明通道。Flash 会读取这些图片并加以处理，然后就能用它们在屏幕上显示带有动画的化身了。

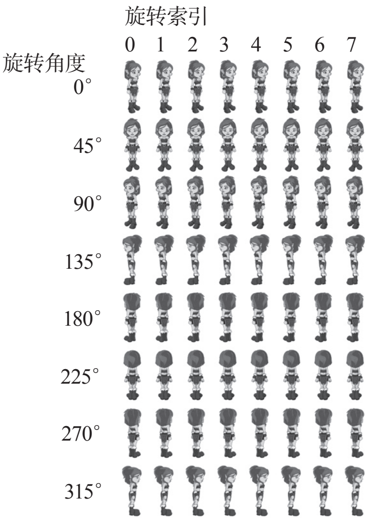


图 13-7 一张精灵序列图中的动画帧

本节我们先学习在客户端显示定制化身的一种方法，首先介绍如何创建精灵序列图。之后再介绍渲染效率和对内存使用的方面。

13.3.1 叠放原则

如何用精灵序列图来显示一个定制的化身呢？你在本章精灵序列图中看到的是一个完整定制的化身角色范例。当用精灵序列图来显示这样一个化身时，你有以下两种方法可选：

- 单独用一张精灵序列图来存储完整定制的化身，或者
- 使用多张精灵序列图，每张图各存储一个定制分类（如用一张精灵序列图来存储衬衣，而另一张存储的是裤子等）。

对于 Flash 开发者来说，第一种方法最好——你只需读取并使用一张精灵序列图即可。但它所带来的潜在问题是：为了记录下所有可能会发生的定制，你必须要在文件系统中创建出所有与之相对应的精灵序列图，只有这样你才能够用一张精灵序列图来存储完整定制的化身。（可以用一些更先进的方法来解决这个问题，但这超出了本书范围，在此不作讨论。）

第二种方法是通过叠放若干精灵序列图来创建一个化身。如用这种方法，你就需要有一个原始的未着装化身。然后将属于不同定制分类的精灵序列图依次叠放在上面。比如，按下面这种由低到高的次序来进行叠放（图 13-8）：

- 人物身体；
- 裤子；
- 鞋子；
- 衬衣；
- 头发。



图 13-8　从本书所用的男性化身的精灵序列图中抽取的单帧图像，按上面范例所用次序进行排列



注意　通常最原始的化身图像还是会带着一点点内衣的，否则会出现裸体化身的 bug（别笑，这种现象确实出现过）。

13.3.2　性能表现

说到精灵序列图的性能表现方面时，我们主要指的是它对帧频（FPS）与内存占用的影响。

在所有我用过的化身绘制法中，精灵序列图技术对帧频的影响是最小的。在 13.4 节中我们会了解相关代码，现在你只需知道的是，精灵序列图会预先被缓存到 BitmapData 实例中，这样就可以使渲染速度变得很快。

一定要随时注意内存的占用，仔细把控，详尽规划。精灵序列图之所以效率高，是由于图像序列都缓存在内存中从而可被快速调取。但是这样做也可能会导致巨大的内存占用。



注意　一张被调用的图像占用多少内存只取决于图像的尺寸，而与图像文件类型和图像压缩无关。同样尺寸下，一个压缩比很大的 JPG 文件和一个细节丰富的 PNG 文件，被 Flash 调取后所占用的内存是一样的。

不过幸好我们可以轻松地计算出任何位图所占用的内存。根据虚拟世界的设定来计算一下，你就能估算出化身会占用多少内存。下面是加载的位图所占用内存的计算公式：

位图所占用的内存（按字节算）＝ 位图宽度 × 位图高度 ×4

用这个等式来算一下，200×400 像素图片所占用内存应为 320 000 字节。除以 1024 转换成千字节则是 312.5 KB。这只是动画一幅单帧的占用。假设化身在 8 个方向上各有一段 12 帧的走路循环动画。那就是 96 帧乘以 312.5 KB，等于 30 000 KB 即 29.3 MB。

让我们将该值用进一法舍入，化身的完整走路循环动画所占用内存就算作 30 MB。如果想在屏幕上同时显示 20 个化身，那就要用掉 600 MB 内存！如果按照很多虚拟世界的做法，你还要给化身再加上一段处于空闲状态时的动画，内存占用就要从 600 MB 翻倍到 1.2 GB 了！天哪！



注意　这里“关键帧”指包含新图像信息的帧。

明白了吧！像这样的估算对前期规划真的很有帮助。要求玩家的电脑配置能让一个 Flash 虚拟世界运行时消耗掉 1.2 GB 的内存？这太不合理了！看看上面的场景设定，你需要赶紧做一些调整来改善这种情况。比如说，你可以把化身的尺寸从 200×400 像素改到 100×200 像素，这样 1.2 GB 就下降到了 293 MB。如果将 12 个关键帧改为 10 个，293 MB 又能降到 244 MB。如果你再把空闲动画去掉的话，那就只需要 122 MB 了。



注意　请记住，内存占用大小不等于你需要下载的文件量大小。我们说化身可能占用 120 MB 内存，但要下载的文件可能只有几兆字节。

怎样做才是最理想的？我认为是这样，整个虚拟世界（化身和其他所有东西）所占用的内存不应超过 350 MB。

创建精灵序列图

因为我推荐把精灵序列图做为绘制化身的好方法，所以你可能想知道如何创建它们。很遗憾，这个问题我也说不好。到现在为止，我们已在好几个项目中用过精灵序列图，每次我们都用自定义代码将单个图片组合在一起生成精灵序列图。而那些单张图片通常是在 Autodesk Maya 里由脚本来控制输出的。

另外，我也听说有一些 3D 软件能够用插件将渲染结果输出为精灵序列图格式。

13.4　创建与定制化身

从现在开始，我们所举的范例都来自“古老家园”这个虚拟世界项目。本章我们只接触“古老家园”项目的一部分，集中讨论如何用精灵序列图来创建与定制化身，以及如何用 ElectroServer 来创建新化身。另外，我们也会学习如何以该化身身份登录以及保存对其的定制。



注意　在 book_files/old_world 里可以找到“古老家园”的文件。本书附录中有关于安装“古老家园”服务器端扩展的更多说明。

创建与定制化身需通过以下 3 个界面（图 13-9）。

- 登录界面（`IntroScreen`）——用户可以在此选择登录或创建一个新化身。
- 注册界面（`RegistrationScreen`）——创建新的化身以及注册相关信息。
- 定制界面（`AvatarCustomizationScreen`）——登录后进行化身定制。



图　13-9

我们还将讨论处理化身与注册系统的常用方法。`IntroScreen` 类没什么新内容好讲的，它只是使登录更方便而已，本书曾讨论过这个类。`RegistrationScreen` 类与 `AvatarCustomizationScreen` 类都会大量使用 `AnimationLoader` 和 `SpriteAnimation` 类，后两个类很重要，它们将有助于你理解如何应用本章所介绍的精灵序列图叠放技术。不过既然两个界面

都会以类似方式使用这两个类，那我们就以 AvatarCustomizationScreen 类为主来讲解如何使用 AnimationLoader 与 SpriteAnimation 这两个类。

13.4.1 概述

“古老家园”支持男女两种类型的化身。两种类型是平等的。男性服装素材只适合男性穿着，反之亦然。一旦被创建，化身就会被存储到服务器端的数据库中。

一个化身有下列属性，大部分都很好理解。

- Name——名称，在创建化身时选定。
- Gender——性别，男或女，也是在创建化身时建立。
- Money——金钱，化身所拥有的货币数量，默认为 1 000 单位的货币。
- Hair——发型，男女两类化身各有两种发型可供选择。
- Top——上身衣服，化身当前所穿的衬衣。
- Bottom——下身衣服，化身当前所穿的裤子或裙子。
- Shoes——鞋。至此，你能看出大体的形象了吧？
- Clothing——包含化身所有服装物品的数组。
- Furniture——包含化身所有家具物品的数组（第 16 章会用到）。

正如你所知道的那样，你必须先登入 ElectroServer 之后才能使用其功能。因为你要使用 ElectroServer 来创建化身，所以你必须在即使没有创建化身时也能登录进去。这种登录类型我们称之为访客登录（guest login）。你先以访客的身份登入并创建一个化身，然后以这个新创建的化身的身份再登录。

登录之后，描述化身可能会拥有的全部服装与家具的数据将作为登录响应从服务器端被加载到客户端。所有这些物品的图像都保存在文件系统里。

Avatar、Clothing 与 Furniture 这 3 个类表示以上我们所讨论的数据实体。AvatarManager、ClothingManager 和 FurnitureManager 这 3 个类则用于轻松地获取或通过 ID 来检索我们所知道的对象。

化身使用货币来购买家具，我们会在第 16 章讲到这一点。

13.4.2 AnimationLoader类和SpriteAnimation类

“古老家园”中的化身形象是由多个精灵序列图叠加构成的。AnimationLoader 类负责加载化身定制所需要的若干精灵序列图，然后将它们按正确顺序添加到 SpriteAnimation 类中。当加载完这些精灵序列图后，我们就不再需要 AnimationLoader 类了，而只使用 SpriteAnimation 类。SpriteAnimation 类得到所有传递来的精灵序列图，将它们合成为一

张总精灵序列图。我们会从这张总图中按照网格顺序抽取所有需要的动画帧，并将它们存储在一个数组中。在缓存完每一个独立的动画帧之后，我们就会将精灵序列图销毁以释放其所占用的内存。



注意　文件出现在数组中的次序决定了叠加的次序。第一个文件位于最低的层次。

我们先来看看 AnimationLoader 类的一个方法，然后再去了解 SpriteAnimation 类的内容。先来创建一个 AnimationLoader 类的新实例，并在其中插入一个 SpriteAnimation 类的实例，然后用 loadFiles 方法读取整个数组。

当所有的文件都加载完毕后，下面这个私有方法就会被调用：

```
private function process():void{
  for (var i:int = 0; i < _loaders.length;++i) {
       var loader:Loader = _loaders[i];
       var b:Bitmap = loader.content as Bitmap;
       _spriteAnimation.layerBitmapData(b.bitmapData);
       b.bitmapData.dispose();
  }
  _spriteAnimation.process();

  _loaders = null;
}
```

每个 Loader 类实例都会加载一张精灵序列图。Loader 类的 content 属性是其所加载的显示对象的引用，在这里就是 Bitmap 实例。接着 Bitmap 实例的 BitmapData 属性被 layerBitmapData 方法（下面会讲解）作为一个图层传递给 _spriteAnimation 实例。然后将此 BitmapData 类实例销毁以释放内存。当处理完所有的 Loader 类实例后，我们就调用 _spriteAnimation 类实例的 process 方法。

现在让我们看一下 SpriteAnimation 类。这里是刚才所用的 layerBitmapData 方法：

```
public function layerBitmapData(bd:BitmapData):void {
  if (_bitmapData == null) {
       _bitmapData = bd.clone();
  } else {
       _bitmapData.draw(bd);
  }
}
```

每加载完一张精灵序列图，该方法就会从 AnimationLoader 类中被调用一次。这里出现了一个 SpriteAnimation 类的类属性 _bitmapData。当其值为 null 时，我们就将传入的位图对象的副本赋予它。如果其值不为 null，那就用 draw 方法将传递来的位图对象叠加绘制在自身之上。BitmapData 的 draw 方法可以选择多种混合模式。混合模式将确定如何合成两个 BitmapData 实例。这里我们使用默认的混合方式，将新的图像直接叠加到原图之上。

当我们创建了一个新的 SpriteAnimation 类的实例后，精灵序列图中的单个动画帧图像的宽度与高度就会被传递给其构造函数。这两个值被赋给 _frameWidth 和 _frameHeight 属性。在 SpriteAnimation 类的 process 方法中将会使用到它们。

```
public function process():void {
  var cols:int = Math.floor(_bitmapData.width / _frameWidth);
  var rows:int = Math.floor(_bitmapData.height / _frameHeight);

  var rect:Rectangle = new Rectangle(0, 0, _frameWidth,
_frameHeight);

  for (var i:int = 0; i < cols;++i) {
       _grid[i] = [];
       for (var j:int = 0; j < rows;++j) {
            var bd:BitmapData = new BitmapData(_frameWidth,
            ➔_frameHeight, true, 0x990000);
            rect.x = i * _frameWidth;
            rect.y = j * _frameHeight;
            bd.copyPixels(_bitmapData, rect, new Point(0, 0));

            _grid[i][j] = bd;
       }
  }

  _bitmapData.dispose();
  _bitmapData = new BitmapData(_frameWidth, _frameHeight, true,
  ➔0x990000);
  nextFrame();
}
```

该方法从合成后的大型 _bitmapData 对象中抽取每一动画帧的图像，并将它们保存到 _grid 数组中。首先，我们要用 _frameWidth 和 _frameHeight 这两个属性来计算出整张精灵序列总图的行列数。接着创建一个新的 Rectangle 实例（马上就会用到），其尺寸与单个动画帧尺寸相同。然后，我们先遍历总图的所有列，为每一列创建一个数组，再遍历每一列中的每一行来处理每一动画帧。在最内层循环体中会创建一个新的 BitmapData 实例 bd，它被用来存储每一个正处理的动画帧的数据。Rectangle 实例被放置与正在处理的动画帧的左上角对齐的位置，这一区域的像素信息就从 _bitmapData 对象被复制到 bd 中。此时包含一个动画帧信息的 bd 再被存储到 _grid 数组中。待处理完 _bitmapData 后，我们会将其从内存中释放，然后为其赋予一个新值。这样当动画播放时，_bitmapData 就会一直指向正确的动画帧。该方法中的最后一步是调用 nextFrame 方法。

```
public function nextFrame():void {
  ++_frameIndex;
  if (_frameIndex == _framesToHold) {
       ++_col;
       _frameIndex = 0;
       if (_col == _grid.length) {
            _col = 0;
       }
```

```
    }
    _bitmapData = getFrame(_col, _row);
}
```

每次此方法被调用时，它都会检测当前应该播放哪一个动画帧。每当创建 SpriteAnimation 类实例时，我们都需要从外部赋值给 _framesToHold 属性。_framesToHold 属性表示的是每一关键帧要保持多少帧。

在“古老家园”中，每一段动画有 8 个关键帧。空闲状态动画每一关键帧保持 6 帧，而行走循环动画每一关键帧则只保持 3 帧。SpriteAnimation 类中的 col 属性指定了要显示动画中的哪一个关键帧。而 row 属性则指定当前帧应处于什么角度（精灵序列图中的每一行都存放着面向不同角度的化身）。

13.4.3　AvatarCustomizationScreen类

当你创建了一个化身并用这个化身登入后，你就会看到化身定制界面。在 GameFlow 类中创建的 AvatarCustomizationScreen 类负责管理这一界面。代表你的化身的一个 Avatar 类实例也在该类中被创建，该类中也创建了一个 Avatar 类，用以代表你的化身，这样该类就获得了一个关于要定制的化身的引用。在此界面你可以浏览所有可选服装。你可以试穿这些服装，化身会实时更新。当你确定更换服装后，服务器会收到一个消息以保存这次修改。

化身由一个 SpriteAnimation 实例绘制而成。我们看一看 buildAvatar 方法中的关键代码。

```
_avatarLoaded = false;

_spriteAnimation = new SpriteAnimation(220, 400);
_spriteAnimation.framesToHold = 6;

_animationLoader = new AnimationLoader();
_animationLoader.spriteAnimation = _spriteAnimation;
_animationLoader.addEventListener(AnimationLoader.DONE,
onAnimationDoneLoading);

var baseDir:String = "files/avatars/big/" + _avatar.gender +
➔ "/";
var urls:Array = [baseDir+"base.png", baseDir+_avatar.bottom.
➔ fileName, baseDir+_avatar.shoes.fileName, baseDir+_avatar.
➔ top.fileName, baseDir+_avatar.hair.fileName];
_animationLoader.loadFiles(urls);
```



注意　buildAvatar 方法在第一次显示定制界面时会被调用，之后在每次服装更换时也会被调用，用以重载化身来显示其更换服装后的外观。

首先将 _avatarLoaded 设为 false。然后创建一个新的 SpriteAnimation 类实例，确

定动画帧的尺寸为 220×400。framesToHold 属性设为 6。再创建一个 AnimationLoader 类实例，为之注册一个 Done 事件的监听器以便当所有加载完成时获得通知。剩下的几行代码创建了一个数组用于存储文件地址，并将其通过 loadFiles 方法传递给 AnimationLoader 类实例。这些地址依照你化身身上的服装生成。

每一帧都会调用 run 方法。

```
private function run(e:Event):void {
  if (_avatarLoaded) {
       _spriteAnimation.nextFrame();
       _avatarBitmap.bitmapData = _spriteAnimation.bitmapData;
       _avatarBitmap.smoothing = true;
  }
}
```

如果全部文件都已加载完毕，我们就会让 SpriteAnimation 实例播放到下一帧，同时将其当前的 BitmapData 对象赋给 _avatarBitmap。由于化身会稍微缩小一点以符合尺寸，所以要把 smoothing 设为 true。

在定制界面还会有一些用户界面元素，你可以通过它们来为化身选择不同的服装。当选择了一件新的服装时，有两件事要处理。

- 调用 save 方法，并传入穿到化身身上的服装的 ID。
- 再次调用 buildAvatar 方法，移除当前的化身并建立一个穿着新服装的化身。

下面是 save 方法。

```
private function save(id:int):void{
  var pr:PluginRequest = new PluginRequest();
  pr.setPluginName("WorldPlugin");

  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.EQUIP);
  esob.setInteger(PluginConstants.CLOTHING_ID, id);

  pr.setEsObject(esob);

  _es.send(pr);
}
```

当你通过用户界面来更改化身的服装时，save 方法就会被调用。该方法接收服装 ID 这个参数，然后用它通过 EQUIP 行为来通知服务器保存为化身所选的服装。

第 14 章

虚 拟 世 界

到目前为止，我们已经讨论了与虚拟世界相关的很多技术，包括时间同步、区块式游戏、A* 寻路算法、等距视图地图绘制以及大量的多人游戏概念。虽然这些概念中的绝大多数都能独立运用，但是本章将把它们结合在一起来演示我们专为本书所创建的虚拟世界——“古老家园”。

在本章中，我们先从共性的角度出发大致介绍一下虚拟世界及其主要技术特点，然后再具体介绍古老家园的特征及其使用的 XML 地图文件格式。 接着我们会关注一下用于渲染生成虚拟世界的一些代码。另外，本章最后将会展示如何将第 13 章中介绍的化身引入虚拟世界中并让它们在其中四处行走。

14.1 共同特征

互联网上有很多 Flash 虚拟世界。如果你创建一个化身并开始在这些虚拟世界中探索，那么你会注意到虽然它们都各有其独特之处，但其中大多数都具有一套共同特征。有些特征虽然不是很普遍，但正变得通用起来，比如像剧情（questing）。

在这一节，我们就来看一下构成虚拟世界的共同特征。

视角——在第 12 章中我们已介绍过视角。虚拟世界的视角会决定所有看到的物体在屏幕上的展示角度，例如俯视角、侧视角、等距视图或者全三维。而在 Flash 虚拟世界中大部分采用等距视图。设计师和开发人员使用等距视图可以作出类似三维的表现力，同时相对于真正的三维方式来说开发周期会短很多。当然，也有少数虚拟世界选择侧视角的方式，如 Whirled（www.whirled.com）。我们假定从现在起讨论的虚拟世界都采用等距视图。

区块式或者绘制式——有些虚拟世界是区块式，有些是绘制式，还有一些虚拟世界两种方式都使用一些。完全使用区块的虚拟世界会将不同种的菱形区块作为填充元素来创建地图，区块式地图中的物品一般会占据一定面积的区块，使得我们很容易实现路径寻找。由绘制式创建而成的虚拟世界一般都采用手绘，其中的物件也都是由手绘而成。一般来说，绘制式虚拟世界只允许化身们直线行走而不能沿任何路径随意行走。

不过把区块式和绘制式结合起来的方式也很普遍，我们的虚拟世界范例“古老家园”就采用了这种方式。在这种混合方式中，物体的摆放和寻路都使用了区块，我们也使用预先绘制好的很大的背景图片。这样做可以表现更加漂亮的背景，同时也可以保留区块式的灵活性。

化身和交互——毫无疑问，化身系统已被所有虚拟世界支持。化身的外观和定制的程度可使虚拟世界各不相同。不同的化身会让玩家通过定制展示个性以及与其他玩家进行互动。

化身间的互动会有很多方式。第一种是聊天和交友。当化身们成为好友时，只要对方在线他们就能够看到，有些时候还可以知道对方在虚拟世界的哪个地方。两种经常出现的化身间交互方式是赠送与交易。赠送，顾名思义，是一个化身送给另一个化身一些东西，经常会是服装之类的东西。交易会使两个玩家互通有无，交易时他们使用交易界面来进行操作。

卷屏——很多 Flash 虚拟世界都用 800×600 像素或者更小的尺寸作为屏幕尺寸，但是玩家进入的虚拟世界一般都比这个尺寸要大。这会用卷屏来控制——当化身在虚拟世界中行走的时候，代码总是会通过卷屏技术来力图使化身接近屏幕中央位置。

卷屏行为可能是最集中考验 Flash 性能的事情了，即使其代码实现起来很简单。有许多的极其先进的技巧和技术手段可以用来优化卷屏，不过这些就不是本书所要讨论的内容了。

室内场景——虚拟世界一般会被认为只是室外场景，但是也有很多虚拟世界允许化身四处走动并找到像房间、建筑或商店这样的结构。这些虚拟世界会允许化身进入这些地方，通常是通过走到指定的区块上或者点击门从而进入。化身离开室外场景后，一个被称作*室内场景*（interior）的地图就加载进来。虽然并没有什么限制，但就一般而言，室内场景的尺寸最多为 1 ～ 2 屏。

常见的室内场景（等距视图下）只能看到背景墙，前景墙、房顶和天花板都看不到。这是为了避免让化身被墙或天花板挡住。

NPC——非玩家角色（NPC）是虚拟世界中不是通过人来控制的各种化身。在多数 Flash 虚拟世界里，NPC 不会四处走动。一般他们都站在一个固定的地方并有一定的用途：比如通过对话提供线索，出售物品或者发布任务（将在后面介绍）。

经济与商家——所有的虚拟世界都会有经济体系。玩家通过现实世界的信用卡换取虚拟货币，或者通过虚拟世界中提供的方式赚取虚拟货币（比如玩游戏）。虚拟货币可以用来购买虚拟世界创造者提供的任何东西（简单地说就是物品），因为玩家喜欢买更多的物品来让化身穿戴或者将它们放在自己虚拟世界的家中。

可以购买物品的地点一般被称为商家，它的外形可能是一个商店，也可能是一个 NPC。

剧情——虽然社交互动是一个人们使用虚拟世界的主要理由，但是有很多虚拟世界都添加了偏向游戏性的内容。剧情可以使玩家在无需跟其他玩家交互的情况下有事可做。剧情由一系列必须完成的任务构成。完成任务和剧情后会给予玩家奖励。奖励可以是任何东西，但是奖励

一般会是虚拟货币或者一件其他任何方式都得不到的物品。

剧情一般是通过和 NPC 交互而提供给玩家。剧情类型有很多，不过一般是下面几种。

- 收集——NPC 可能告诉你收集 20 个苹果给他。
- 递送——NPC 可能告诉让你把东西送给指定的地点或 NPC 处。
- 访问——让你去某个地方，到了那个地方就算完成任务。

剧情系统可以为虚拟世界添加一整套新的游戏和活动方式，这些特征虽不是很普遍，但现在也开始越来越多地出现在虚拟世界中。

游戏——在许多 Flash 虚拟世界中，除了定制自己的化身和房间之外，玩家的主要活动当然还是玩游戏。有些游戏是单人的而有些是多人的。绝大多数虚拟世界都会在某些地方设置进入这些游戏的入口。

游戏的进入一般是通过虚拟世界的 NPC 或者走到了某个特定位置。游戏都比较简单，可以玩上几秒到几分钟，结束后玩家会获得虚拟货币或者其他奖励。

用户之家——很多虚拟世界都会给化身一个虚拟的家，经常是一个 1 屏大小的地方。化身可使用储物栏里现有物品来装饰定制自己的家。这些物品可能是购买的，或是剧情奖励和玩游戏赢来的。



注意　*化身的好友可以访问住宅，在第 16 章中我们会详述用户之家。*

地图编辑器——地图编辑器通常是用来让开发者创建并维护虚拟世界的，对于玩家来说它毫无意义。一个虚拟世界会有很多区域，几十个甚至数百个。地图编辑器是一种定制的工具，能够让你很方便地设定预设资源、区块或者背景图并把它们可视化地展现出来。开发一个这样的编辑工具是设计虚拟世界所必须的——因为通过手写 XML 来定义虚拟世界的布局是一件非常耗时的工作，所以花两到三天来开发一个地图编辑器将会节省数倍的时间，而这些时间通常是以后用来创建和编辑地图的。

14.2 “古老家园”

正如在第 13 章提到过的那样，我们专为本书创建了一个叫做“古老家园”的虚拟世界范例。古老家园（图 14-1）具备了虚拟世界的很多基本特征。它使用了等距视图技术，而且混合使用了区块式与绘制式方法。虽然背景图像是预先绘制而成的，但是我们依然采用区块来布置虚拟世界中的物品，并且将其用于行走和寻路的计算之中。

“古老家园”中有室内场景和室外场景。我们可以从外部环境中进入室内环境。图 14-2 是一个卖家具的 NPC 所住的室内房间。



图 14-1 欢迎进入“古老家园”



图 14-2 Nara 遇到了一个在“古老家园”中卖家具的 NPC

家具购买后就可以在用户之家中使用了（在第 16 章中会介绍）。在用户之家中，你可以将买来的家具随意摆放并将配置保存起来（见图 14-3）。

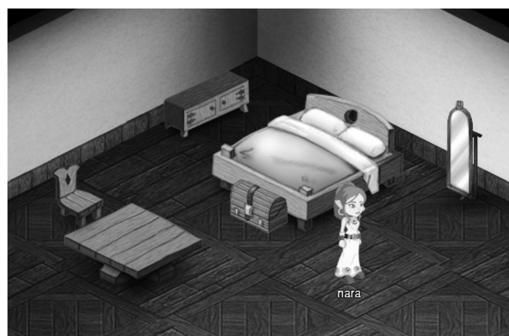


图 14-3

你可以聊天，还可以通过点击其他化身向其发出好友请求（图 14-4）。你能在任何时候查看好友列表，看看他们是否在线，同时还可以访问他们的用户之家。

你可以离开虚拟世界去定制化身，然后再进入虚拟世界。



图 14-4



注意 化身的定制界面与其代码已在第 13 章介绍过。

“古老家园”也有一个地图编辑器，它可以让你轻松且直观地创建新地图。

“古老家园”中暂时还没有以下功能（并不是说这些功能不能被加入）。

- ❒ 整合游戏。在几个小时内就可以为“古老家园”植入游戏，但现在这里没有任何游戏。
- ❒ 剧情。
- ❒ 交易与馈赠。
- ❒ 更高级的房间定制功能，比如可以加入小地毯、壁纸的颜色与图案、挂画或窗户，以及可旋转方位的物件。（不过像摆放椅子、床和桌子这样的基本房间布局定制在“古老家园”中都已经有了。）

14.3 地图文件

古老家园中每一区域的布局都是通过（或者说是“在”）一个 XML 文件定义的。XML 文件可以用像记事本这样的文本编辑器来手动创建，也可以通过“古老家园”的地图编辑器来输出。本节就让我们来查看一下 XML 地图文件格式详细的定义，然后再简要地了解一下地图编辑器。

14.3.1 XML格式

由于“古老家园”只是一个阐述了基本功能的简单虚拟世界，所以它的 XML 格式比较简单。而且在后面说明这些格式的全部功能时，我们会尽量少介绍 XML 相关知识。为了方便后面的引用，代码行前都加了行号。

```
1.    <map>
2.         <background file="floor2.png" x_offset="-510"
           ➔ y_offset="-5" cols="15" rows="15"/>
3.         <ItemDefinitions>
4.              <ItemDefinition id="chair2_a" file="chair2_a.
                ➔ png" x_offset="-40" y_offset="-40"
                ➔ rows="1" cols="1" walkable="true"
                ➔ overlap="false"/>
5.              <ItemDefinition id="nightstand1_a" file=
                ➔ "nightstand1_a.png" x_offset="-75"
```

```
                  ➔y_offset="-25" rows="2" cols="1" walkable=
                  ➔"true" overlap="false"/>
6.          </ItemDefinitions>
7.          <Items>
8.               <Item source="chair2_a" col="2" row="1"/>
9.               <Item source="chair2_a" col="1" row="2"/>
10.              <Item source="nightstand1_a" col="3"
                 ➔row="7"/>
11.         </Items>
12.         <Tiles>
13.              <Tile col="4" row="2" walkability="false"
                 ➔placeability="false"/>
14.              <Tile col="7" row="3" walkability="true"
                 ➔placeability="false"/>
15.              <Tile col="11" row="4" walkability="false"
                 ➔placeability="true"/>
16.         </Tiles>
17.    </map>
```

1. **BackGround** 节点

第 2 行中有一个 <background> 节点，它定义了地图所使用的背景图片文件。和该 XML 文件中定义的所有图片一样，它的格式既可以是 PNG 也可以是 JPG。<background> 节点的参数如下。

- file——需要加载图片的文件名。这个文件必须在相对于 Oldworld.swf 文件的 /assets 目录里。
- x_offset——该参数用于定义图片在屏幕坐标系 *x* 轴上的偏移量。通过使用 x_ffset 与 y_offset 可以使等距视图的（0，0）点能够与图片的正确位置对齐。
- y_offset——跟 x_offset 共同使用。
- cols——地图占据的区块列数。
- rows——地图占据的区块行数。

2. **ItemDefinitions** 与 **ItemDefinition** 节点

<ItemDefinitions> 节点包含着尽可能多的所需的 <ItemDefinition> 节点。（是的！这确实是两种不同的节点名称。）每一个 <ItemDefinition> 节点都定义了一个能出现在虚拟世界中的物件的全部信息。例如，同样一棵树需要在虚拟世界中被使用 50 次。那么我们只需使用 <ItemDefinition> 对如何生成这棵树定义一次，然后再使用 <Item> 节点（下面会介绍）来放置它的实例即可。在上述 XML 代码中的第 4 行和第 5 行有 <ItemDefinition> 节点的范例。其中一个说明椅子只占用了一个区块，另一个说明床头柜占用了两个区块。

每个 <ItemDefinition> 节点都有如下参数。

- id——<Item> 节点使用该参数来指定到底使用哪个 <ItemDefinition> 节点。

- file——图片文件的位置。
- x_offset——和 y_offset 一起使用，用来使图像相对区块的位置进行偏移，以保证物件能够准确定位。
- y_offset——和 x_offset 一起使用。
- cols——表示物件占用区块的列数。
- rows——表示物体占用区块的行数。
- walkable——布尔值。如果其值为 true，并且该物件被放置到一个 walkable 属性也为 true 的区块上的话，化身就可以在该区块上行走。
- overlap——布尔值。如果为 true, 那么该物件就可以与其他物件并存于同一区块上。

3. Items 与 Item 节点

<Items> 节点包含着尽可能多的所需的 <Item> 节点，以便显示所有被放置到虚拟世界中的物件。<Item> 节点定义了 <ItemDefinition> 的一个实例，同时指定它应该被放置到哪个区块上。范例中的第 8 行和第 9 行中的两个 <Item> 节点指定使用了同一个物件定义 ID。

每个 <Item> 节点都有如下参数。

- source——字符串值，它对应着 <ItemDefinition> 节点中的 id 参数。
- col——表示放置物件的区块的列序数。
- row——表示放置物件的区块的行序数。
- onStop——可选参数。如果该参数存在，那么其值通常为需要加载的另一个 XML 文件的路径。如果化身停留在这个物件上，那么就会加载一个新的区域。
- onclick——可选参数。如果该参数存在，那么当玩家点击物件时将产生点击事件，并且该参数值将被调入到这个点击事件中。这个参数通常用来实现玩家与物件的互动，比如与 NPC 的互动。

4. Tiles 与 Tile 节点

当地图文件被加载进“古老家园”之后，它将创建一些区块，其数量为 <Background> 节点中的 rows 值与 cols 值的乘积。如果有 10 行 10 列，那么将有 100 个区块存在内存中。这些区块的默认值允许在其上放置物体并允许化身在上面行走。

这部分 XML 代码能让你确定不宜设置默认值的特定区块。<Tiles> 节点包含了尽可能多的 <Tile> 节点，以用来指定所有特定区块。范例中的第 13 行、第 14 行和第 15 行代码分别重载了 3 个不同区块的默认属性。每一个 <Tile> 节点都具有下列参数。

- col——指定区块的列序数。
- row——指定区块的行序数。
- walkability——布尔值。默认值为 true，如果为 false，表示化身不能在其上行走。

- `placeability`——布尔值。如果为 false，表示其他物件不能被放置在该区块上。该参数在定义用户之家的 XML 格式时非常有用。

14.3.2　地图编辑器

我们为“古老家园”开发了一个基本的地图编辑器。它是一个 Adobe AIR 程序。



注意　地图编辑器可在 book_files/old_world/editor 目录中找到。双击即可安装。如果你还没有安装 Adobe AIR，可以从 http://get.adobe.com/air/ 处下载并安装。

安装完编辑器后，运行它。然后选择 Options → Change Source Directory。你会被提示要选择一个目录。你所选择的目录应该包含一个 assets 文件夹和一个 data 文件夹，就像“古老家园”源文件中的 bin 目录一样（图 14-5）。data 文件夹应该包含一个名为 Asset List.xml 的文件。编辑器通过该文件来确定能被放置在地图上的物件。



图　14-5

如想添加一个物件到虚拟世界中，只需从图 14-6 左侧的列表中将它拖曳出来然后放到地图上即可。如果想移动物件，可通过点击来将它拾取，然后再点击地图上的其他位置来将它放置。

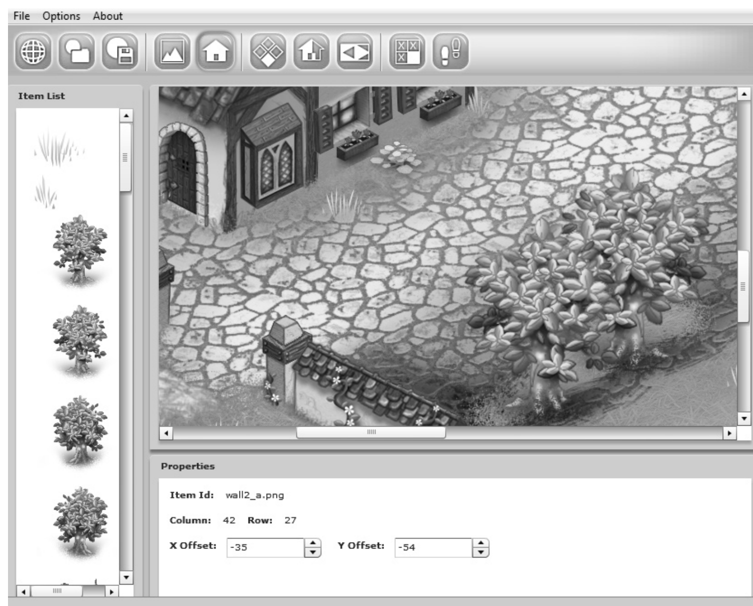


图 14-6　你可以在编辑器窗口的底端面板中调整物件的偏移量

编辑器的限制

使用这个编辑器，你基本上可以完成创建一个完整的虚拟世界所需做的一切。但是它不具备以下功能。

- 指定一个能够传送的物件。
- 指定一个能够交互的物件，例如 NPC。

当你准备要保存地图时，只需点击 Save 按钮即可。

14.4 地图的渲染生成

在第 12 章和第 13 章中，我们介绍过区块与等距视图的概念，并介绍了如何使用 `Isometric` 类与排序算法在屏幕上布置物件。在本节中，我们来看一下在“古老家园”中如何运用我们已经学过的概念来创建并渲染场景。

14.4.1 Map类

为了渲染生成一张地图，“古老家园”需要加载一个地图 XML 文件，将其解析后加载该 XML 文件中所有定义的物件，接着绘制地图背景并把物件放置其上。这些功能都是通过 `Map` 类来完成的。

`Map` 类用 `loadMap` 方法来加载地图 XML 文件，当所有物件被加载并渲染生成之后，`Map` 类就触发一个 `Map.READY` 事件。`Map` 类中还有一个叫 `isEditable` 的布尔值属性，其默认值为 `false`。如果将其设置为 `true`，那么 `Map` 类允许从类外部添加、删除及移动物件。`Map` 类中提供了这样处理物件的各种方法与事件（第 16 章会讨论）。



注意 在第 12 章我们讨论过排序的问题，但是我们在本章稍后部分将会介绍它是如何被用来处理动态运动中的化身的。眼下，我们只需要知道排序是在 `Map` 类中处理的。

14.4.2 Isortable接口

所有需要在虚拟世界中排序的物件（这里指的就是 `Item` 类与 `Avatar` 类）都必须实现 `ISortable` 接口。如要实现该接口，只需具备下列属性即可。

- `col`——物件所在区块的列序数。
- `row`——物件所在区块的行序数。
- `cols`——物件所占用区块的列数。

- rows——物件所占用区块的行数。

排序算法与我们在第 12 章介绍的一样。但是关于何时排序以及如何管理排序列表将会有稍许不同，本章稍后我们都会讲到。

14.4.3 ItemDefinition类

前面我们介绍了 XML 文件中的 <ItemDefinition> 节点。而 ItemDefinition 类就是该节点的 ActionScript 同等表述。该类存储了 walkable 和 overlap 属性，以及物件所占用区块的行列数。

除此之外，ItemDefinition 类还包括了图片信息。图片通过 ItemManager 实例载入后，BitmapData 实例就保存在对应的 ItemDefinition 实例中。

14.4.4 Item类

地图 XML 文件中包含一个 <Item> 节点，同样，Item 类就是该节点的 ActionScript 的同等表述。每一个 Item 类实例都会对应一个要使用的 ItemDefinition 对象以及其在地图中被放置的位置。由于 ItemDefinition 对象包含着代表可视化物件的 BitmapData 对象，所以所有的 Item 实例都使用同样的 BitmapData 对象。这样可以维持低内存占用率。例如，如果某个 ItemDefinition 对象表示一棵树，并且它在地图中被使用了 100 次，那么这颗树的图片在内存中仍然只存在一次。

Item 类中有一个很有用的方法叫 chekPointCollision，它可以用来确定某点是否正和物件的不透明区域接触。如果你想通过点击鼠标与一个物件交互，那么使用通常的鼠标点击事件是不可行的。因为正常的鼠标点击事件是即使当鼠标点击到物件完全透明区域也会被触发的。然而，checkPointCollsion 方法可以帮助我们解决这个问题，它会只对物件的非透明区域触发鼠标点击事件。

```
public function checkPointCollision(tx:int, ty:int):Boolean {
  var collision:Boolean =
_bitmap.bitmapData.getPixel32(tx - _bitmap.x, ty - _bitmap.y)
➔ != 0;
    return collision;
}
```

该方法通过使用调入的点与 BitmapData 实例共同来检测该点所经过的像素的 ARGB 值，如果像素的 ARGB 值为 0 则表示它是透明的，也就是说交互点不在图片的可见区域。

14.4.5 ItemManager类

ItemManager 类加载所有在 ItemDefinition 对象中指定的图片文件，然后将 Item 实例

跟正确的 ItemDefinition 实例结合起来。此类还会将所有的 Item 实例和 ItemDefinition 实例按照它们的 ID 保存在数组中，以方便查找。

14.5　虚拟世界

在第 13 章中，我们介绍过定制化身的程序。在该定制程序界面上有一个 Enter World（进入虚拟世界）按钮。点击之后定制界面就消失了，你的化身就进入了虚拟世界的室外环境当中，如图 14-7 所示。



图　14-7

当你点击 Enter World 按钮时，GameFlow 类就会调用下列函数：

```
private function onEnterWorldClicked(e:Event):void {
  removeAvatarCustomizationScreen();
  createWorld("data/world.xml");
}
```

它先移除了化身定制界面，然后调用 createWorld 方法，该方法负责创建你要进入的虚拟世界场景。

```
private function createWorld(url:String, home:Boolean=false,
➔ owner:String=null):void{
  _world = new World();
  _world.addEventListener(World.TELEPORT, onTeleport);
  _world.addEventListener(World.GO_TO_HOME, onGoToHome);
  _world.es = _es;
  _world.clock = _clock;
```

```
    _world.clothingManager = _clothingManager;
    _world.furnitureManager = _furnitureManager;
    _world.buddyList = _buddyList;
    _world.initialize(url, home, owner);

    addChild(_world);
  }
```

该方法有 3 个参数。第一个参数为 url，它指定一个你要加入的虚拟世界的 XML 文件的路径。第二个参数为 home，默认为 false；如果为 true，则表示所创建的虚拟世界场景是用户之家，并且规定了第三个参数必须被赋值。第三个参数为 owner，它指明了拥有这个刚加入的用户之家的化身名称。

该方法首先创建了一个新的 World 类实例。早先我们已经介绍了 Map 类以及它是如何渲染生成虚拟世界的。World 类实例封装了 Map 类实例，并且会处理所有的用户界面代码以及客户端与服务器端间的通信。然后我们添加了两个事件监听器，用来处理当化身走到传送点时或者当玩家点击用户界面上的按钮返回自己家时所触发的事件。

接下来的几行代码使 World 类可以使用 GameFlow 类所记录下的一些数据，它们中的大多数都在前面的章节中讨论过。_buddyList 属性是一个 AvatarManager 实例，它有一个包含你所有好友的列表（我们将在第 15 章介绍它）。而 _world.initialize 方法会在传递 createWorld 函数该接受的 3 个属性（url、home 与 owner）的同时被调用。

让我们跳转到 World 类来讲一下 initialize 方法。当所有只能从外部设置的属性（比如 ElectroServer 实例）都被设置好之后，该方法会被 GameFlow 类调用。这个方法会创建一个新的 Map 实例，并监听在其上触发的一些事件，然后让 Map 实例加载地图。同时它还会创建一个 Astar 实例（用于寻路），并添加用户界面元素。

当地图被完整加载后，onMapReady 事件处理器就会被触发，它能够依次把玩家添加到一个房间中。这里的代码很好理解，如下所示。

```
private function joinRoom():void{
  var crr:CreateRoomRequest = new CreateRoomRequest();
  crr.setRoomName(_mapUrl);
  crr.setZoneName("world zone");

  var pl:Plugin = new Plugin();
  pl.setPluginHandle("AreaPlugin");
  pl.setPluginName("AreaPlugin");
  pl.setExtensionName("GameBook");

  if (_isHome) {
       crr.setRoomName(_owner);
       pl.getData().setString(PluginConstants.ROOM_OWNER,
       ➔ _owner);
  }
```

```
    crr.setPlugins([pl]);

    _es.send(crr);
}
```

进入到虚拟世界相当于加入一个房间。使用地图的 URL 作为房间名是为了方便将玩家们组织起来。请注意 CreateRoomRequest 对象默认情况下会将你加入到你指定名称的房间（如果它存在的话），或者创建一个新房间（如果它不存在）。所以，通过将地图 URL 作为房间名称，你最终会和那些使用同样地图的其他玩家在同一个房间中。服务器上的一个插件会和该房间关联，这样它就可以控制化身列表与他们的行走事件了。如果你加入的房间是一个用户之家，那么你加入的房间名称会变为房屋主人的名称，并将该名称传到该插件中。

就像本书前面的范例一样，当 JoinRoomEvent 事件产生时，客户端会发送一个 INIT_ME 消息到插件。而插件就会识别出客户端并将其作为虚拟世界的一部分，并且会给该客户端发送化身列表，以及所发生的化身添加 / 移除事件以及化身行走事件。

14.5.1　化身管理

进入房间后的客户端会接收到一个化身列表。列表中包含了所有在虚拟世界中的现存化身的信息，比如它们身上的服装以及它们最近的行走路线。这些事件与虚拟世界中的所有服务器事件都被 World 类接收并解析。根据需要，World 类会调用 Map 类的 addAvatar 方法和 removeAvatar 方法。

addAvatar 方法会添加一个化身到 AvatarManager 实例中。通过将这个化身的引用调入到 placeSortableItem 方法中，该化身也会以可排序显示对象的形式被添加进来（我们会在 14.5.3 节讨论此点）。所有可排序物件都会被添加到同一个叫做 _ground 的显示对象中。有些物件应该始终保持在顶层，比如聊天气泡以及化身名片。化身名片是一个文本框，位于化身的脚下，如图 14-8 所示。聊天气泡和化身名片作为子对象被添加在 _toplayer 显示对象中，_toplayer 显示对象始终在 _gournd 显示对象的顶层。

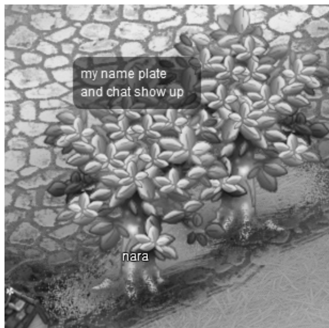


图 14-8　化身在树的后面，但是聊天气泡和名片则显示在树的前面

removeAvatar 方法只将化身名称作为参数。它通过名称找到该化身，然后将该化身以显示对象的形式从 AvatarManager 实例中移除。同时它也会移除聊天气泡字段和该化身的名片。

14.5.2 行走

让化身在屏幕中行走并不难，但是这需要用到我们在书中学过的很多概念，比如服务器通信、A* 算法、基于时间的运动、化身精灵序列图绘制法。我们先来介绍基本流程，然后再来看一些实现代码。

首先我们会监听地图上的点击事件。如果一个区块被点击，则 Map 类会触发一个事件，并且该事件会被 World 类所捕捉。我们在从化身当前所处区块到你所点击的区块之间执行一个 A* 搜索。所产生的路径会被格式化，接着它会随同从 _clock 实例取得的时间标签一齐被发送到服务器端。游戏中的所有玩家（包括你）都会接收到你的化身的行走事件。路径被解析为 Tile 实例，然后被作为 WayPoint 类实例封装起来（后面会讨论）。然后我们通过基于时间的运动方式让你的化身逐步运动。化身就会播放行走动画并且会在运动时显示正确的转向。

下面的 World 类的事件处理器代码用来处理区块点击事件。

```
private function onTileClicked(e:TileEvent):void {
  var startNode:INode = _map.getTile(_map.avatarManager.me.col,
  ➔ _map.avatarManager.me.row);
  var goalNode:INode = e.tile;

  var results:SearchResults = _astar.search(startNode,
  ➔ goalNode);
  if (results.getIsSuccess()) {
       sendWalkPath(results.getPath());
  }
}
```

所有的 Tile 类实例都实现了 INode 接口，因此它们可以与 A* 算法协同工作。StartNode 是指化身当前位置，而 goalNode 则是指化身需要走到的位置，也就是我们点击的区块的位置。搜索过程中我们会通过 _astar Astar 类实例来使用这两个节点。如果寻路成功，那么 Path 实例会被传递给 sendWalkPath 方法并格式化，然后与一个时间标签一起被发送到服务器端。

当行走事件被客户端接收后，它会被 World 类捕捉并解析。World 类会调用 Map 类的 walkAvatar 方法。

```
public function walkAvatar(name:String, time:Number,
➔ tiles:Array):void {
  var avatar:Avatar = _avatarManager.avatarByName(name);
  var wpIndex:int = 0;
  var wayPoints:Array = [];

  for (var i:int = 0; i < tiles.length;++i) {
```

```
            var tile:Tile = tiles[i];
            var dis:Number;
            if (i != 0) {
                 dis = getDistance(tile, tiles[i - 1]);
                 time += dis / avatar.walkSpeed;
            }

            var wp:WayPoint = new WayPoint();
            wp.time = Math.round(time);
            wp.tile = tile;

            if (_clock.time > wp.time) {
                 wpIndex = i;
            }
            wayPoints.push(wp);
       }
       avatar.walk(wayPoints);
       avatar.wayPointIndex = wpIndex;
  }
```

walkAvatar 方法有几个参数：化身名称、开始行走时的时间，以及一个组成该路径的 Tile 实例数组。该方法的作用是创建 WayPoint 类实例，并使每一个实例对应着路径上的每一个点。WayPoint 类封装了 Tile 实例的引用与化身到达该 Tile 实例引用的时间。我们通过遍历 Tile 实例数组创建一个 WayPoint 类实例数组。对于我们所关注的每一个区块，我们都会计算出其前一个区块与它的距离，同时根据化身行走速度来计算出化身何时会到达该区块。Tile 类实例和时间一起被传入一个新的 WayPoint 实例中，然后该实例就会被添加到数组 WayPoints 中。由于行走是基于时间的，而我们无法控制网络延时，并且一个新玩家可能会在另一个化身走在半道上时加入进来，所以我们将 WayPoint 时间与服务器端当前时间进行对比，以此来检查化身当前应处于哪个路点上。在代码的最后两行，我们将 WayPoints 数组传入到化身的 walk 方法中，从而使化身能够沿着指定路线行走，然后设置当前的 wayPointIndex，该参数指定了化身当前所处的路点。

在每一帧中，Map 类中 moveAvatars 方法都会被调用。

```
private function moveAvatars():void{
  for each (var avatar:Avatar in _avatarManager.avatars) {
       if (avatar.state == Avatar.WALKING) {
            stepAvatar(avatar);
       }
       avatar.run();
  }
}
```

每个化身都会处于两种可能的状态之一：空闲或者行走。在这个方法里，我们会遍历所有化身。如果化身当前是行走状态，那么我们会将它传递到 stepAvatar 方法中，我们马上就会讲到该方法。不管是空闲状态还是行走状态，我们都会调用化身的 run 方法。该方法会确定化身的正确方向、播放空闲动画或者行走动画的下一帧。同时，它也会让化身的名片与聊天气泡

对应到正确的相对位置上。

这里是 stepAvatar 方法的代码：

```
private function stepAvatar(avatar:Avatar):void{
  var time:Number = _clock.time;
  var wp:WayPoint;
  var ind:int = avatar.wayPointIndex;

  // 是时候进入下一个路点了吗？
  if (ind < avatar.wayPoints.length - 1) {
       wp = avatar.wayPoints[ind + 1];
       if (time > wp.time) {
            avatar.wayPointIndex = ind + 1;
            ind = ind +1;
       }
  }

  // 当前路点
  wp = avatar.wayPoints[ind];

  var x:Number;
  var y:Number = 0;
  var z:Number;

  // 在等距空间中的位置
  x = _tileWidth * wp.tile.col;
  z = _tileHeight * wp.tile.row;

  var elapsed:Number = _clock.time - wp.time;

  if (ind == avatar.wayPoints.length - 1) {
       avatar.changeState(Avatar.IDLE);
       checkForOnStopEvent();
  } else {
       x += elapsed * avatar.walkSpeed * avatar.cosAngle;
       z += elapsed * avatar.walkSpeed * avatar.sinAngle;
  }

  var coord:Coordinate = _iso.mapToScreen(x, y, -z);
  avatar.x = coord.x;
  avatar.y = coord.y;
}
```

任何行走的化身在每一帧中都会执行 stepAvatar 方法，它将根据从化身到达当前路点后所经历的时间，使化身沿着路径从当前路点运动到下一个路点。如果时间推移得足够长，我们将把下一路点改变为当前路点。X 和 Z 变量被初始化以用来表示当前路点的位置。然后我们还要给这两个数值加上根据自从到达当前路点后所经历时间而算出的数值。随即我们会使用 Isometric 类实例的 mapToScreen 方法将这些坐标映射到屏幕上，并且更新化身在屏幕上的 x 与 y 坐标。

14.5.3 排序

在第 12 章介绍过的排序对比逻辑也用在该项目中。不过在这里，用法会稍微有些差别。在

虚拟世界里既有静态物体又有运动物体。其中会有成百个像是树木、灌木和房屋这样的静态物体。其中可能只有少量的一些运动物体——在我们这个虚拟世界中指的就是化身们。静态物体只需进行一次排序，因为它们彼此间不会运动。在这个项目里我们将可排序的静态物体保存在一个数组中，然后每一帧我们只要将化身排序，插入到数组中。

我们在将可排序物体（如化身或者房子）添加到屏幕上时会调用 placeSortableItem 方法。

```
private function placeSortableItem(sortable:ISortable,
➔ insert:Boolean = true):void{
  // 添加到显示列表中
  _ground.addChild(sortable as DisplayObject);

  if (insert) {
       // 添加到可排序的项目列表中
       _sortables.push(sortable);
  }

  // 取得 3D 坐标
  var iso_x:Number = sortable.col * _tileWidth;
  var iso_z:Number = -sortable.row * _tileHeight;

  // 把 3D 坐标映射到屏幕上
  var screenCoord:Coordinate = _iso.mapToScreen(iso_x, 0,
  ➔ iso_z);

  // 基于屏幕坐标更新显示对象
  (sortable as DisplayObject).x = screenCoord.x;
  (sortable as DisplayObject).y = screenCoord.y;
}
```

所有可排序物体必须实现 ISortable 接口。如果你对第 12 章还有印象的话，那么你应该记得这意味着它必须提供参数 col、row、cols、rows。如果 placeSortableItem 方法的 insert 参数为 true，那么这个物体会添加到 _sortables 数组。当化身实例传递到这个方法后，insert 会被设置为 false。以上代码中的后几行会把该化身放置到屏幕上正确的区块中。

另外，在每一帧中都会调用 sortMovableItems 方法。

```
private function sortMovableItems():void {
  var list:Array = _sortables.slice(0);
  var moving_arr:Array = _avatarManager.avatars;

  for (var i:int = 0; i < moving_arr.length;++i) {
       var nsi:ISortable = moving_arr[i];
       var added:Boolean = false;
       for (var j:int = 0; j < list.length;++j ) {
            var si:ISortable = list[j];

            if (nsi.col <= si.col + si.cols - 1 && nsi.row <=
            ➔ si.row + si.rows - 1) {
                 list.splice(j, 0, nsi);
                 added = true;
```

```
                    break;
                }
            }
            if (!added) {
                list.push(nsi);
            }
        }

        for (i = 0; i < list.length;++i) {
            var disp:DisplayObject = list[i] as DisplayObject;
            _ground.addChildAt(disp, i);
        }
    }
```

sortMovableItems 方法的用处是将所有化身按排列好的顺序插入静态物体数组中，然后该静态物体数组会被复制到一个新数组 list 中。我们先创建了一个包含所有化身的局部变量 moving_arr。我们会循环访问 moving_arr 数组并使用排序逻辑来确定化身应该插入到 list 数组中的哪一个位置。最后我们会遍历 list 数组将每一个已排序显示对象添加到场景的正确索引中。

购买物品

一个在“古老家园”的旅店里卖家具的 NPC（见图 14-9 至图 14-11）。



图 14-9　若想进入旅馆，你只需走到前门处的传送点即可



图 14-10　点击 NPC 能看到让你浏览待售商品的用户界面元素

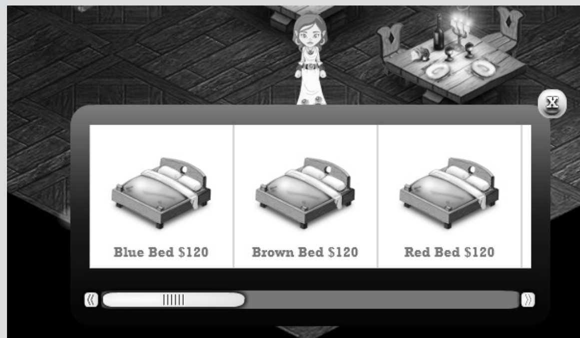


图 14-11　你在“古老家园”中买不到服装，只能买到家具



注意　因为只有当你在自己家中时才能使用你的家具物件，所以只能当你进入自己的家时，它们才会被加载进来。我们会在第 16 章中介绍更多这方面的内容。

你可以卷动查看可买的家具并点击想买的那一种。如果你确定了购买，BUY_ITEM 请求会被发送到服务器。

在 FurnitureManager 实例中能找到那些可买物品。它们是作为登录响应的一部分被加载的。

第 15 章

好 友 系 统

虚拟世界就是社会化网络。它们之所以如此受人追捧，在很大程度上是由于化身间能够互动并建立各种关系。化身间能建立的最普遍关系是好友，也就是通常说的*朋友*。一旦你与另一个化身建立起某种关系，你就能通过虚拟世界所提供的各种途径直接与该化身互动，无论通过私密聊天、一起玩游戏，还是通过其他独特的交流方式。

本章中，我们将

- ❑ 分析何为关系，并说明几种类型的关系；
- ❑ 探讨一下建立关系所能用的不同方法；
- ❑ 访问那些存在于“古老家园”（Old World）中的好友们，并浏览下相关的代码。

15.1 关系

社会化网络的基础在于能够让人与人之间建立联系，并且相互间有一定程度的交互。从核心理念上来讲，像 Facebook、MySpace 以及 Old World 这样的程序都非常地相似——它们提供了一种新颖的方式让人们之间彼此相识，从而建立起某种关系，并允许建立起关系的人之间相互交流。

15.1.1 关系类型

我之所以把人们之间的联系称作*关系*而不是通常意义上的朋友或好友，是由于关系的多样性。所存在的关系类型是没有任何限度的。下面就是我在一些虚拟世界或其他社会化网络中所看到的类型。

1. 朋友 / 好友

这是最常见的一种关系，它允许玩家把其他玩家添加到其好友列表中。在他们被添加到好友列表之后，如何与其互动则依赖于你所在虚拟世界的处理方式。不过，一般情况下你至少可以做到下面几点：

- ❑ 显示其是否在线；

- 向其发送私密消息；
- 知道其在虚拟世界中的位置。

在许多虚拟世界中，你还可以对好友们这么做。

- 向他们发出游戏挑战（或邀请）。
- 赠送给他们礼物。礼物通常是储物栏的某种物品，比如帽子或衬衫。
- 给他们发送虚拟世界内部的电子邮件。这是给离线好友发送消息的一种方式。如果邮件还支持附件的话，这将会是赠送物品的另一种方式。

2. 忽略 / 屏蔽

这听起来似乎不像是关系，但它确实是一种特殊的关系。玩家之间可相互交流也意味着给他们提供了辱骂诽谤的机会。辱骂诽谤可以有不同的形式，但通常都发生在聊天时。如果一个玩家辱骂或者用其他手段骚扰了其他玩家，那么受害者就可以将他添加到忽略名单中。玩家是不会收到其忽略名单中的玩家所发来的聊天消息或者其他事件的。

3. 父母 / 家庭

大多数 Flash 虚拟世界的目标群体是 8 ～ 12 岁的未成年人。由于安全问题备受关注，有些虚拟世界开始允许家长查看孩子们的活动信息，甚至控制孩子们访问虚拟世界的次数。特殊的家长与孩子之间的关系就是为此而建立的。

在某些虚拟世界中，家庭成员跟朋友一样都是同等对待的，只是添加了一个额外参数以表明该成员是家庭成员。

4. 合作者 / 同事

这种关系在虚拟世界中很少看到，不过，在像 Facebook 和 LinkedIn 这样的社会化网络中这种关系却很常见。那些在一起工作的职场人员间可以建立一种关系。通常，关于他们的共同工作年限以及各人技能的相关信息会与关系一同保存。

5. 敌人 / 仇敌

社会化网络是虚拟世界中的一个重点，但不是唯一。许多虚拟世界正在添加进游戏元素，比如剧情任务或者战斗，如此玩家除了社会化互动就有其他事可做了。敌人是一种在拥有竞争性游戏元素的虚拟世界中所存在的关系类别——敌人只是相对于游戏环境而言，它并非指你实际上讨厌的人。

15.1.2　建立关系

在虚拟世界中，关系通常是这样被创建的：当你在虚拟世界中看到其他人的化身时，通过

右击该化身便会弹出一个上下文菜单，然后选择一个选项来创建一种关系（例如把他加为好友或者丢进黑名单）。但是这仅仅只是你的一厢情愿，并不意味着你就能实现这种关系。每种关系都有两个属性，它们定义了关系是如何被建立的。

1. 对称性

关系的对称性是指是否双方都能同意建立此关系。举个范例，如果玩家 A 忽略了玩家 B，这并不意味了玩家 B 也必须忽略玩家 A。忽略是一种非对称的关系。但在大多数情况中，好友关系都是对称性的：如果玩家 A 将玩家 B 加为好友，那么玩家 B 同样也会将玩家 A 加为好友 (见图 15-1)。

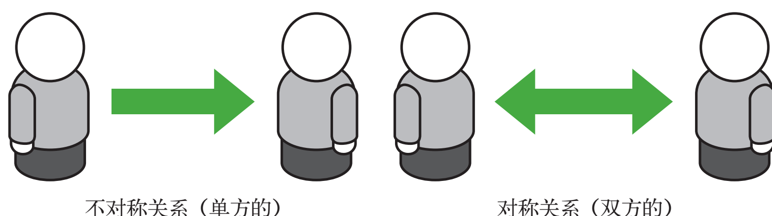


图 15-1

2. 授权

关系授权可以用很多不同方法来处理 , 我们在这里介绍最常见的一种方式。当玩家 A 尝试与玩家 B 建立某种关系时，玩家 B 就会得知此节并予以验证，然后接受或拒绝建立这种关系。如果关系被接受，那么双方都会被告知该关系已经建立 ; 如果关系被拒绝，那么玩家 A 就会知道他的请求被拒绝，如此一来双方没有任何关系可言。

大多数情况下，对称关系都是需要授权的，而非对称关系则不需要。例如，当玩家 A 想忽略玩家 B 时，他并不需要玩家 B 的授权。

15.2 “古老家园”中的好友

“古老家园”的好友系统非常简单（图 15-2）。为了易于编程与说明，好友关系被设计成非对称的。后面我们将解释“古老家园”好友系统的所有功能，并且会介绍好友关系内部的一些代码。



图 15-2

ElectroServer 内置了好友概念。虚拟世界的客户端可使用 WorldPlugin 插件从数据库中添加和删除好友。当一个玩家登录后但未进入虚拟世界时，他能从 WorldPlugin 插件中完整地加载好友列表。这只需加载一次。该列表包含了玩家所有好友的名字与其所对应的化身的 ID，并且还包含一个布尔值来表明他们是否在线。从这时

起，你就可使用名为 BuddyStatusUpdatedEvent 的 ElectroServer 事件去更新好友的在线 / 离线状态了。当某个好友登录或注销时，这个事件就会被触发。由于这是 ElectroServer 内置功能，所以 ElectroServer 知道好友何时登录或注销。当 WorldPlugin 插件为某玩家加载其好友列表时，它会在内存中将每一个化身都注册为好友，从而能够让 ElectroServer 在正确的时间触发 BuddyStatusUpdatedEvent 事件。

15.2.1 加载好友列表

让我们来重述一下前面段落的内容，当玩家登录后但还未进入虚拟世界时，他就加载了好友列表。这些是在 GameFlow 类中完成的。onLoginResponse 事件处理器会调用下面的函数：

```
private function loadBuddies():void {
  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔ LOAD_BUDDIES);

  sendToWorldPlugin(esob);
}
```

我们用 LOAD_BUDDIES 行为来格式化一个 EsObject 对象。当 WorldPlugin 插件接收到该请求后会为那个化身加载好友列表，并最终回应一个 LOAD_BUDDIES 响应，而这些都是在 GameFlow 类的 handleLoadBuddies 方法中得到处理的。

```
private function handleLoadBuddies(esob:EsObject):void{
  var list:Array = esob.getEsObjectArray(PluginConstants.
  ➔ BUDDY_LIST);
  for each (var buddyOb:EsObject in list) {
       var avatar:Avatar = new Avatar();
       avatar.avatarName = buddyOb.getString(PluginConstants.
       ➔ BUDDY_NAME);
       avatar.avatarId = buddyOb.getInteger(PluginConstants.
       ➔ BUDDY_ID);
       avatar.isOnline = buddyOb.getBoolean(PluginConstants.
       ➔ LOGGED_IN);

       _buddyList.addAvatar(avatar);
  }
}
```

好友列表是由一个 AvatarManager 类实例来管理的，该实例名为（且慢，你准备好了吗？）_buddyList。在上面的函数中，我们对从服务器发送过来的代表好友的 EsObject 对象数组进行遍历，依次将它们解析成 Avatar 类实例，然后再把它们添加到 _buddyList 中。无论什么时候创建 World 实例，_buddyList 实例都会在其中被创建。这就是 World 实例能够访问好友列表并能将其显示在用户界面上的原因。



注意 每一位好友都有 3 个属性：名字、ID，以及一个表明其是否在线的布尔值。

15.2.2 显示在线好友

好友列表的用户界面将在下节中讨论，我们用一个颜色指示器来表明好友是否在线。如果好友（一个 Avatar 类实例）的 isOnline 属性为 true，那就表明好友在线并且其颜色指示器显示为黄色。当好友列表加载之后这个属性值就会被初始化。不过随着时间的推移，好友们会不断地登陆与注销，而我们希望 isOnline 能始终保持最新状态。为了做到这一点，我们需要监听名为 BuddyStatusUpdatedEvent 的 ElectroServer 事件。当好友登录或注销时，该事件就会被触发。这个事件处理程序在 GameFlow 类中，如下所示。

```
public function onBuddyStatusUpdatedEvent
➥ (e:BuddyStatusUpdatedEvent):void {
  var name:String = e.getUserName();
  var isOnline:Boolean = e.getActionId() ==
  ➥ BuddyStatusUpdatedEvent.LoggedIn;

  if (_buddyList.avatarByName(name) != null) {
       _buddyList.avatarByName(name).isOnline = isOnline;
  }
}
```

该事件处理器从 _buddyList 中找出正确的好友，然后根据事件中的 getActionId 方法的返回值来更新 isOnline 属性值。假如 getActionId 方法的返回值与 BuddyStatusUpdatedEvent.LoggedIn 属性值相同，那就说明该好友已经登录；否则，就说明该好友已经注销了。

15.2.3 添加好友

涉及添加好友的大部分代码都是有关用户界面的。我们将讨论一下用于驱动用户界面的相关函数，不过代码这块只看一看有关 ElectroServer 如何添加好友的部分。

当你在虚拟世界中看到另一个化身时，你可以点击它。World 类中的 onAvatarClicked 方法将处理该次点击。只要你没有对自己点击，你都将会看到一个弹出式对话框（见图 15-3），询问你是否要把该化身添加为好友。



图 15-3

如果你点击 No 按钮，该弹出式对话框就会消失。如果你点击 Yes 按钮，就会调用 onBuddyConfirmYes 事件处理器。该方法将向 ElectroServer 发送请求以添加好友，并直接把好友添加到好友列表中，然后移除弹出式对话框。

```
private function onBuddyConfirmYes(e:Event):void {
  var pop:BuddyConfirmationPopup = e.target as
  ➥ BuddyConfirmationPopup;
```

```
    var avatar:Avatar = pop.avatar;

    var esob:EsObject = new EsObject();
    esob.setString(PluginConstants.ACTION, PluginConstants.
    ➥ ADD_BUDDY);
    esob.setInteger(PluginConstants.BUDDY_ID, avatar.avatarId);

    sendToWorldPlugin(esob);

    _buddyList.addAvatar(avatar);

    removeBuddyConfirmPopup(pop);
  }
```

先从弹出对话框中获取化身的引用。接着我们用一个 ADD_BUDDY 请求来格式化一个 EsObject 对象并将其发送给 WorldPlugin 插件。然后我们直接把该好友添加到好友列表中。随后从屏幕上移除弹出式对话框 UI 元素。

15.2.4　移除好友

移除好友是在好友列表中进行的。如果你选择了好友列表中的一位好友，那么有两个按钮会被激活，即 Remove 和 Home（图 15-4）。



图　15-4

如果你点击了 Remove 按钮，选中好友即被删除。下面就是点击 Remove 按钮后，World 类调用的函数。

```
  private function removeBuddy(avatar:Avatar):void {
    var esob:EsObject = new EsObject();
    esob.setString(PluginConstants.ACTION, PluginConstants.
    ➥ REMOVE_BUDDY);
    esob.setInteger(PluginConstants.BUDDY_ID, avatar.avatarId);

    sendToWorldPlugin(esob);

    _buddyList.removeAvatar(avatar.avatarName);
  }
```

我们用 REMOVE_BUDDY 行为来格式化一个 EsObject 对象，然后将它发送给 WorldPlugin 插件，插件随后会将该好友从数据库中移除，并且通过 ElectroServer 将该好友注销掉。该函数的最后一行直接从好友列表中移除了该好友的化身。

15.2.5　查看好友列表

查看好友列表使用了简单的用户界面代码。屏幕右下方带有桃心图标的按钮即是好友列表的按钮。

点击好友列表按钮，World 类中的 onBuddyListClicked 事件处理器就会被调用，该事

件处理器接着会调用 createBuddyListUI 函数。createBuddyListUI 函数会创建一个新的 BuddyList 类实例（图 15-5）。这个 BuddyList 类就是用来显示好友列表的显示对象。



图 15-5

当创建了 BuddyList 类实例后，一个 Avatar 类实例数组就会传给它以显示好友。在好友名字的左边会有一个带颜色的圆圈以表示该好友是否在线。红色表示离线，黄色表示在线。如果你选择了列表中的一位好友，就像在 15.2.4 节中所学的那样，Remove 按钮与 Home 按钮就会变成启用状态。一旦你点击了 Remove 按钮，该好友就会从好友列表中移除并且会向 WorldPlugin 插件发送一个 REMOVE_BUDDY 请求。通过点击 Home 按钮，你将会被带到该好友的用户之家。第 16 章中我们将介绍用户之家。

如果你点击了用户列表中的 X 按钮，那么好友列表将会从屏幕上移除。

15.2.6 可改进之处

如要在一个对公众开放的虚拟世界中使用好友系统，则你需要增加一些额外的功能以使其变得更有用。

- **对称性添加好友**——在这个范例中，你可以把任何人添加为好友而无需对方同意，并且在他们手动将你加为好友之前，你是不会成为他们的好友的。我们可以予以改进，使得你所选择的化身在你将其加为好友时能够对你进行验证，虽然麻烦，但是通过验证，你们各自都将成为对方的好友，就像在其他虚拟世界中所做的那样。
- **显示好友位置**——我们当前只能显示好友是否在线。如果你能够确切地知道好友在虚拟世界中的位置，或者你能直接被传送到他们所处位置，则将有助于查找好友并与之互动。
- **私密聊天**——要是有了这个常用的功能，你就能和好友私密交谈了。它能通过好友列表而得以初始化。服务器端已能完整地支持该功能，所以你无需修改任何代码。你唯一需要做的就是在客户端上添加用户界面元素以使其支持该功能。

由于玩家间的互动是用户能否逗留并重返虚拟世界的首要因素之一，所以你有必要花些时间来开发多种功能以支持这些互动。

第16章

用户之家

如前所述，虚拟世界就是社会化网络。它们的核心价值就在于能够实现玩家间的互动并展现玩家的不同个性。我们已经概述了在虚拟世界中玩家间如何互动并确立各种关系，以及玩家如何通过定制他们的化身来彰显个性。除此之外，大多数虚拟世界还有一个用来展示玩家自我风格的常见功能——用户之家。

化身的用户之家就是玩家在虚拟世界中的私有空间，玩家可以用化身储物栏中的物品来定制它。化身也可以邀请好友到自己的房间里，顺便显摆一下他们的物品，这些物品有的是他们购买的，有的是完成某个任务的奖励，有的是玩游戏所得。与定制好的化身有所不同，无论其“主人”是否在线，用户之家总能被其他人访问。

本章中，我们首先来介绍可能与用户之家有关的所有内容，随后讨论为“古老家园”而开发的一简单的用户之家功能。

16.1 “打开房间”

本节中，我们将概述一下用户之家的常见特性。就充当一回房客吧，如果你喜欢这样的话，到处转转，仔细看看。用户之家几乎具备了虚拟世界的所有特点：化身能够进入用户之家、在其中四处走动、与其他化身进行互动。不同点在于用户之家还具有一些在通常的虚拟世界环境中所没有的功能。大多数的这些功能所关心的是如何使用户之家拥有者对房间进行定制（图16-1）。

在绝大多数虚拟世界中，每一玩家都拥有单独的用户之家。他们可以对自己的房间进行定制，比如可以添加或删除一些物品、改变墙壁的颜色或图案，甚至还可以改变房间自身的布局（房间尺寸以及墙壁的方位）。

你还可以进入其他玩家的家。但在是否实现以及如何实现该功能上，不同虚拟世界间有所不同。通常玩家只能访问其好友的家。而另一种情况也很常见，即好友不必在线或在其房间，别人就能访问他的家。但有些虚拟世界则明确规定，当主人不在线或不在家的情况下，其他玩家是不能访问的。



图 16-1 地毯、窗帘、壁画……可定制的房间布置可使用户之家新鲜每一天

用户之家中的物品

用户之家中的大部分装饰是通过物品的摆设来完成的。在虚拟世界中，化身获取物品的方式有多种多样：购买、完成任务所得、朋友的赠送或者交易。大部分获得的物品不但可以打扮化身自身，还可以用来装饰他们的房间。

用户之家可使用的物品可归结为如下几类。

- **家具物品**——这种物品是用户之家中最常见的环境物品了。它包括诸如椅子、桌子、橱柜和床这样的物品。
- **墙壁物品**——这种物品只能用在墙上。常见的墙壁物品有海报、挂钟、门和窗。
- **地面物品**——这种物品只能在地板上使用，并且你能够将家具物品放置其上。毯子就是一种常见的地面物品（见图 16-2）。

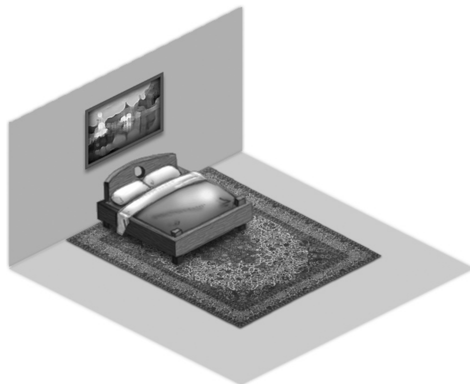


图 16-2 一个有家具、墙壁、地面物品的范例



注意　通常你应该为物品创建出多种可见的旋转角度。这样，玩家在虚拟世界中放置该物品时就能选择该用哪个角度来进行放置。

玩家能够直观地将上述物品摆放到虚拟世界中。通常，玩家能够在储物栏中看到他所拥有的所有物品。于是玩家就能从用户界面中拖动出物品并将其放置到自己家中。物品在被拖动时所展现出的运动特点与其类型有关（图16-3）。比如，家具物品通常会在拖动时与鼠标保持位置一致，然而墙壁物品则会在拖动时沿着墙运动。

至此，本章所提到的这些物品与定制功能可以带给玩家很大的灵活性来使自己的家更具个性。此外，还有一些高级的定制功能能用来提高定制的灵活性。下面就是一些物品可具有的额外功能。

- **状态**——物品可以做成能显示不同的状态。比如说，房间里的灯能够开启或者关闭。
- **动画**——物品不必都像椅子那样是静态图片。还可以通过添加一些动画来使其具有生机感。比如，鱼缸可以添加进鱼儿畅游的动画。
- **可堆叠性**——可堆叠物品指的就是那些能够放到其他物品之上的东西（图16-4）。比如像电视、台灯、花瓶、相框之类的东西。一些物品只能被设计成可堆叠物品，而有些物品只能被设计成能够承载可堆叠物品。比如，你不能将某件物品放到咖啡杯上面，但你也许会将咖啡杯放到床头柜上面。
- **自动移动**——这种物品熟知用户之家的布局，它能根据定制的逻辑来引导自己在房间中运动。它可能会是任何东西，比如像遥控汽车或Roomba牌免提式真空吸尘器之类的东西。

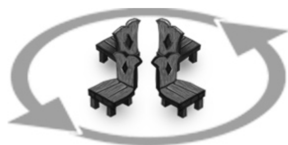


图　16-3

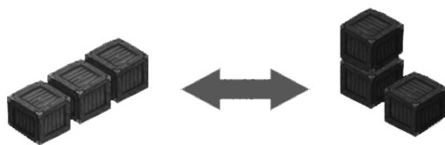


图　16-4

用户之家的布局

在对化身的家进行定制的过程中，虽然物品的作用非常重要，但周围环境的变更也能带来很大的改观。你可以试着将地面的材质由地毯变为实木地板，或者变换一下墙壁的颜色，比如说由亮紫色改为粉绿色，这些改变都能使用户在定制自己房间的过程中获得另外的用于展现自我个性的途径。

刚开始时，玩家可能会满足于住在不超过一屏的小房间中。但正如现实生活中那样，随着时间推移，他们也希望来点改变——一般都是要求升级。不同的房间结构会包含着不同的墙壁布局及地板铺排方式，而这些都能使玩家重拾起对家的兴趣。

16.2 “古老家园”中的用户之家

现在你已经熟悉了用户之家及其很多特点。下面让我们来看看“古老家园”中已编程实现了的用户之家范例吧。

16.2.1 访问与装饰

在这个范例中，你可以进入另一个玩家的家，当然，他们必须是你的好友才行。打开好友列表，然后选择其中的一个好友。你就会看到 Home 按钮已经被激活了（图 16-5）。点击这个按钮，你就能进入该好友的家（图 16-6）。

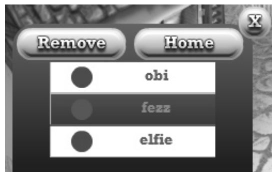


图 16-5



图 16-6

要想进入你自己的家，只需点击屏幕右下方的 Home 按钮即可。

一旦你身处自己家中，你就可以对它进行定制。你的家可以处于这样两种状态之一：设计状态或者正常状态。当它处于设计状态时，你就可以布置它但你却无法四处走动，而当它处于正常状态时则正好相反。

想要定制你自己的家，只需点击屏幕右下方的铁锤图按钮即可。

当你进入设计模式后，你会看见一个用户界面，它包含你所拥有的尚未布置在房间里的全部家具物品。如果你还一件东西都没有的话，你可以去旅馆（参见第 14 章的内容）找 NPC 去购买。想要把某件物品从用户界面拖到你自己的家中，只需单击它一次。然后，你会发现，当你运动鼠标时，那个物品也紧随其后。在家里选好合适的放置点，再点击一次，该物品就布置好了。

如要移动某件物品，单击它将其拾取，然后移动鼠标在屏幕上移动它，再次单击就可以放置它了。要移除某件物品，先拾取该物品，而后运动鼠标至 Recycle 按钮上并点击（图 16-7），那么该物品就会从你家中移除并返回到储物栏。

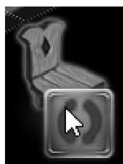


图16-7　不用担心，当你“回收”某个物品时，它并不是真的被删除了，而只是回到了储物栏

现在，你已对“古老家园”中用户之家的操作有了感性的认识，让我们更进一步，从技术角度看看这些操作都是如何实现的吧。

16.2.2　数据与事务处理

在这一节中，我们来了解一下客户端是如何知道所有已存在的家具的，以及当化身进入其家中时，客户端又是如何知道该化身所拥有的家具的。此外，我们还将介绍当某个物品发生位移时，客户端又是如何通知服务器端的。

首先，在GameFlow类所处理的登录响应中，还有额外的信息被传送给客户端。在第13章中，我们曾经知道登录响应对象包含着与衣服有关的信息，它能被parseClothing方法所解析。在传递给onLoginResponse事件处理器的EsObject对象中还能发现与家具有关的信息。虚拟世界中所有的家具（不一定是化身所属的）都在此时被传送给客户端。然后parseFurnitureItems方法就会将其解析。我们稍后再介绍该方法，在这之前，首先让我们复习一下与家具信息有关的一些知识，并介绍一些新的内容。

首先，让我们来快速温习一下Furniture、FurnitureDefinition、Item与ItemDefinition类：Item与ItemDefinition类都在第14章已经介绍过。你大概还会想起ItemDefinition类包含了一个物件以及其相对于区块偏移量的可视化信息。Item类被用于包含ItemDefinition类实例的位置信息。比如，你可以有20个Item实例，它们分布在不同的位置上，但都使用着同一个ItemDefinition类实例。

除此之外，你也应该知道Furniture类会为化身所拥有的每一个家具物品都建立一个实例。Furniture类实例含有指向Item类实例的指针，我们用Item类实例将家具物件摆放到虚拟世界中，另外在Furniture类实例中还含有一个FurnitureDefinition类实例。FurnitureDefinition保存着已经购买的家具物件的信息，比如该家具物件的文件名、花费以及类型ID。

卖家具的NPC用一张包含所有FurnitureDefinition对象的列表来显示适合出售的物件。玩家能够放置到房间中的储物栏物件全都是Furniture类实例。

现在我们全都明白了，让我们再回到parseFurnitureItems方法上来，该方法会在GameFlow类中的onLoginResponse事件处理器中被调用：

```
private function parseFurnitureItems(items:Array):void{
  for (var i:int = 0; i < items.length;++i) {
       var furni:FurnitureDefinition =
       ➛ new FurnitureDefinition();

       var furniOb:EsObject = items[i];
       furni.name = furniOb.getString(PluginConstants.
       ➛ FURNITURE_NAME);
       furni.fileName = furniOb.getString(PluginConstants.
       ➛ FURNITURE_FILE_NAME);
       furni.id = furniOb.getInteger(PluginConstants.
       ➛ FURNITURE_ID);
       furni.cost = furniOb.getInteger(PluginConstants.
       ➛ FURNITURE_COST);

       _furnitureManager.addFurnitureDefinition(furni);
  }
}
```

这是一个系统中所有待售家具物件的列表。调入该方法中的是一个 EsObject 对象数组。每个 EsObject 对象所包含的信息先是被解析为 FurnitureDefinition 实例，然后被存储进 FurnitureManager 类的一个对象 _furnitureManager 中。

剩余的用户之家的客户端与服务器端间事务是在 World 类中处理的。在第 14 章我们讲过，当化身进入一个新的区域时，首先会创建一个新的 World 类的实例来管理用户界面，然后创建一个 Map 类实例，接着加载 XML 布局以及所有的图片。在 World 类的 initialize 方法中有一个类属性叫做 _isHome，它是一个布尔值属性，如果化身进入的是一个房间的话，那么其值为 ture。我们马上就能用到这个属性，同样在 16.2.3 节也要用到它。

在第 14 章中我们还讲过当化身刚加入一个区域后，该区域的全部化身的列表就会马上被发送给客户端。World 类中的 handleAvatarList 方法会处理这个列表。下面是这个方法末尾的一个 if 语句，我们当时介绍该方法的时候没有讲它，现在让我们看一下。

```
if (_isHome) {
  parseHomeFurniture(esob);
}
```

如果化身所加入的区域是用户之家，那么发送给客户端的 EsObject 对象同样也包含客户端拥有的全部家具以及它们的位置信息。parseHomeFurniture 方法会处理这些对象。该方法会将 EsObject 对象解析为 Furniture 实例，并且为了方便访问它们，还会用以下 3 种方法对其进行保存：将其用一个叫做 _furniture 的典型索引数组保存起来；将其按照 ID 保存在一个叫做 _furnitureByEntryId 的 Dictionary 类实例中；将其按照 Item 类实例保存在另一个叫做 _furnitureByItem 的 Dictionary 类实例中。用这些不同的方式来保存 Furniture 类实例，这样以后我们就能轻松地访问这些数据了。比如，如果在虚拟世界中点击了一个物件，我们就会用 Item 类实例来查找 Furniture 类实例。

最后我们要介绍的客户端与服务器端通信的问题就是关于运动物件的。一个物件可能会被从用户界面移动到虚拟世界中，也可能会被从虚拟世界的某处移动到另一处，或者会被从虚拟世界再移回到用户界面。不管以上这 3 种情况何时发生，都会调用下面的方法：

```
private function moveFurniture(item:Item, inWorld:Boolean):
➔ void {
  var furni:Furniture = _furnitureByItem[item];

  var esob:EsObject = new EsObject();
  esob.setString(PluginConstants.ACTION, PluginConstants.
  ➔ MOVE_FURNITURE);
  esob.setInteger(PluginConstants.FURNITURE_ENTRY_ID, furni.
  ➔ entryId);
  esob.setInteger(PluginConstants.PLACEMENT_ROW, item.row);
  esob.setInteger(PluginConstants.PLACEMENT_COLUMN, item.col);
  esob.setBoolean(PluginConstants.FURNITURE_IN_WORLD, inWorld);

  sendToAreaPlugin(esob);
}
```

该方法有两个参数，即 `item` 与 `inWorld`。`item` 参数含有被移动的 `Item` 类实例。`inWorld` 参数是一个用于确定物件是否在虚拟世界中的布尔值。如果 `inWorld` 为 `true`，那么 `Item` 类实例就在虚拟世界中；反之，`Item` 实例就回到了储物栏里。

该方法的剩余部分将物件信息格式化为 EsObject 对象，然后再将其发送到服务器端。被发送的信息有：被移动物件的 ID、物件的行列位置以及用于确定物件是否在虚拟世界中的布尔值。

总的来说，所有的 `FurnitureDefinitions` 都被作为登录响应的一部分而被加载进来，当你进入一个玩家的家时，该玩家拥有的全部 `Furniture` 类实例都将提供给你，并且当一件家具被移动到新位置上时，`moveFurniture` 方法就会被用来将该信息告知给服务器端。

16.2.3 用户界面

在本节中，我们来看看一些用来驱动用户界面的代码，比如我们何时允许客户端编辑房间而何时不允许，又比如我们如何通过用户之家的选择窗口将一件还没有被放置的物件从储物栏中运动到房间的某处。看看下面的代码（取自 `GameFlow` 类中的 `createWorld` 方法）：

```
_world.initialize(url, home, owner);
```

当 `Wolrd` 类实例创建好后，如你刚才所见，其中的 `initialize` 方法就会被调用。`url` 参数指向一个用来定义静态虚拟世界布局的 XML 文件。home 参数是一个布尔值，如果其为 `true`，则化身进入的是用户之家。`owner` 属性包含一个字符串值，它只有当 `home` 为 `true` 时才有效，如果 `home` 为 `true`，它就包含着拥有该住宅的化身的名称。

下面是 `World` 类中 `initialize` 方法的头几行代码：

```
public function initialize(url:String, home:Boolean,
➔ owner:String):void {
  _mapUrl = url;
  _isHome = home;
  _owner = owner;
  _isMyHome = _es.getUserManager().getMe().getUserName() ==
  ➔ owner;
```

你可以看出，该方法引用了调入的参数。除此之外，如果 `owner` 值与正运行该代码的客户端的化身名称相同的话，`_isMyHome` 就会被设为 true。`_isMyHome` 属性用来确定客户端是否能够编辑这个指定房间。这是在 `World` 类中的 `onMapReady` 事件处理器中进行的。让我们现在来看一下：

```
if (_isMyHome) {

  // Alter Bottom UI "Home Buttom" => World Button
  _bottomUI.home_btn.visible = false;
  _bottomUI.worldButton.visible = true;
  _bottomUI.addEventListener(UserHomesEvent.EXIT_HOMES,
  ➔ onHomesExited);

  // Add Bottom UI for Homes
  _homesBottomUI = new HomesUI();
  _homesBottomUI.addEventListener(UserHomesEvent.EDIT_MODE_
  ➔ TOGGLE, onEditModeToggle);
  _homesBottomUI.addEventListener(UserHomesEvent.ITEM_RECYCLED,
  ➔ onItemRecycled);
  _homesBottomUI.x = 687.5;
  _homesBottomUI.y = 550;
  addChild(_homesBottomUI);

  // Create Furniture Selection UI
  _furnitureList = new UserHomesItemList();
  _furnitureList.addEventListener(UserHomesEvent.ITEM_SELECTED,
  ➔ onListItemSelected);
  _furnitureList.visible = false;
  addChild(_furnitureList);

  // Add Listeners to Map for in-world Item Interaction
  _map.addEventListener(ItemInteractionEvent.ITEM_SELECTED,
  ➔ onItemSelected);
  _map.addEventListener(ItemInteractionEvent.ITEM_PLACED,
  ➔ onItemPlaced);
}
```

如果 `_isMyHome` 为 true，那么我们就会用上述方法在屏幕上添加指定的用户之家界面。如你所见，这会改变许多状态：Home 按钮被设为不可见，Globe 按钮被设为可见并且会监听离开家的事件。另外，还会有两个额外的按钮被添加到屏幕右下角的按钮列表中。其中锤状的按钮控制的是用户之家的普通模式与定制模式的切换。另一个（最左边的那个）按钮用来将物件回收到储物栏中。接下来会创建一个 `UserHomesItemList` 类的新实例，并将其存储为

_furnitureList。这个用户界面元素展示了一张还没有被放置到虚拟世界中的所有家具物件的列表。代码的最后两行为 Map 类实例添加了事件监听器，它们是用来捕捉这样的事件：当某个虚拟世界中的物件被选中时，以及当某个物件被放置到地图上时。

一开始，我们将 _furnitureList 的 visible 属性设为 false，因为用户之家默认情况下是普通模式。一旦玩家点击锤状图标进入定制模式，_furnitureList 就会显示出来，玩家的家具就会显示在该窗口上（图 16-8）。



图　16-8

一旦我们选中了 _furnitureList 中的一个物件，onListItemSelected 事件处理器就会被调用：

```
private function onListItemSelected(e:UserHomesEvent):void {
  _map.startDraggingItem(e.item);
  e.item.filters = [new GlowFilter(0x009900)];
}
```

被选中的物件存在于调入的事件对象中，并随即传给 Map 类实例的 startDraggingItem 方法。调用该方法能够将物件可视化地添加到地图中，并将其顺序排在其他所有物件之上，而且该物件还能随着鼠标的运动而运动。物件四周还会产生发光效果以表明它被选择了。

当物件实际地被放置到虚拟世界中时，onItemPlaced 事件处理器就会被调用：

```
private function onItemPlaced(e:ItemInteractionEvent):void {
  e.item.filters = [];
  moveFurniture(e.item, true);
}
```

该事件处理器会将物件的选择状态时的滤镜效果移除，并且调用 moveFurniture 方法（本章早先讨论过）。它会与服务器端进行通信，通过更新数据库来保存该物件的新位置。

如果玩家决定将该物件放回到储物栏中，他们可以将其拾取并点击 Recycle 按钮。当 Recycle 按钮被选择时，onItemRecycled 事件处理器就会被调用：

```
private function onItemRecycled(e:UserHomesEvent = null):void {
  var item:Item = _map.itemBeingDragged;
```

```
    if (!item) {
         return;
    }
    if (_furnitureList) {
         _furnitureList.add(item);
    }
    _map.stopDraggingItem();
    moveFurniture(item, false);
}
```

该事件处理器取得正被拖动的物件，将其重新添加到 _furnitureList 类实例中，并告知 Map 类实例停止对它拖动，然后调用 moveFurniture 方法来将该物件的新位置储存到数据库中。

好了！我们已经介绍了所有跟处理客户端与服务器端通信有关的代码，而且也知道了该怎么样显示及放置物件。如果你想深化一下这个范例，那么最好下一步为其添加物件旋转功能。但那取决于你——现在就看你的了。祝你玩得开心！

附录

创建范例扩展包

在你下载的与本书配套的范例文件 zip 压缩包里，包含了一个扩展包（extension），该扩展包可以直接部署到 ElectroServer 的安装目录中，而无需重新编译 java 文件。在 book_files/examples_extension/extension 文件夹下，找到并复制 GameBook 文件夹。然后将它粘贴到 ES4 安装文件夹下的 folder/server/extensions 中。重新启动 ES4，然后添加服务器端（server-level）的组件。

A.1 服务器端组件

完成扩展包部署之后，重启 ElectroServer 并打开管理面板。如果是在本地运行 ElectroServer，则只需要打开 Web 浏览器，然后导向 https://localhost:8080/admin。接着以管理员身份登录[①]，进入 Extensions 选项卡。你将在这个界面中看到 ElectroServer 中安装的所有扩展包。点击加号按钮（+），然后点击 New Server-Level Component 按钮。下一个界面显示的就是 GameBook 下的扩展包。从 Plugin Handle 的下拉列表中选择 Time StampPlugin（图 A-1），并且在插件名（Plugin Name）的输入字段中输入 TimeStampPlugin。

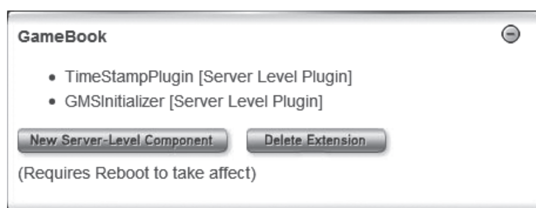


图　A-1

按照以上步骤添加 GMSInitializer 插件，并再次重启 ElectroServer。



注意　你也可以使用一个与下拉列表项不同的插件名字，但是如果两者一致的话将更方便进行问题追踪。

① 登录名和密码参见 4.2.1 节。——译者注

A.2 “古老家园”

由于 OldWorld 需要登录验证，而上述的 GameBook 扩展包内未包含 OldWorld 内容。你下载的范例 zip 压缩包内有单独的 OldWorld 文件夹，你可以依照配置其他范例一样，在 ElectroServer 中部署针对 OldWorld 的扩展包。找到 book_files/old_world/server_extension/oldWorldExtension 目录，复制其中的 GameBook 文件夹，然后粘贴到 ES4 安装目录的 folder/server/extensions 下。你也可以把两个 GameBook 扩展包[①]合并为一个，只需把两个扩展包中内容放到一起。重启 ES4，然后添加新的服务器级（server-level）组件。这些步骤都完成以后，你所设置的服务器级组件应该包含这些项目（图 A-2）：

- GMSInitializer（对于“OldWorld”不是必要的）；
- TimeStampPlugin；
- WorldPlugin；
- LoginEventHandler；
- LogoutEventHandler。

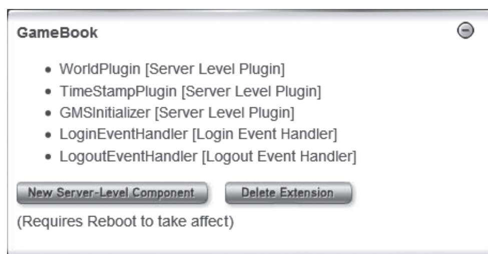


图　A-2

你还需要为 OldWorld 部署数据库。找到 book_files/old_world/server_extension/server/db 目录，并复制其中的 BookWorld 文件夹到 ES4 安装目录下的 folder/server/db。

再一次重启 ES4，此时 OldWorld 就可以运行了。

A.3 配置日志

如果你想设定服务器的日志为 Debug 模式，以便更容易地进行问题诊断，则需要找到 ES4 安装目录下的 folder/server/config 目录，编辑其中的 log4j.properties 文件，在其中添加下面两行：

```
log4j.logger.com.gamebook=debug
log4j.logger.Extensions.GameBook=debug
```

① 上面的 GameBook 和 OldWorld 的这个 GameBook。——译者注

重启 Electroserver 之后，你应该可以在服务器日志中看到每一条插件信息了。请注意这在产品正式运营期间是不需要的，但是在开发阶段是非常有用的。

A.4　创建服务器端开发环境

如果你需要对提供的 Java 源代码进行修改，或者期望编写自己的服务端插件，那么附录的剩余部分将是为你而准备的。

A.4.1　安装Java 和 NetBeans

Java 环境和 NetBeans 都是免费下载的。其他的编辑器也能够支持 Java，而本附录假设你正在使用 NetBeans。

从 http://java.sun.com/ 下载并安装最新版本的 Java SE Development Kit（JDK）。如果你手头上没有与 Java 捆绑发行的 NetBeans，你可以在 www.netbeans.org 下载到。

A.4.2　创建NetBeans项目

完成 Java 环境和 NetBeans 的安装之后，为 GameBook 扩展包创建 NetBeans 项目将变得轻而易举。

(1) 复制 book_files/examples_extension/server 文件夹下的 Java 源码。我们假设你把它复制到 C:/GameBook/server。

(2) 打开 NetBeans，然后选择 File → New Project 创建一个项目。

(3) 在 Categories list（分类列表）中选择 Java。在 Projects list（项目列表）中选择 Java Free-Form 项目。点击 Next 按钮（图 A-3）。

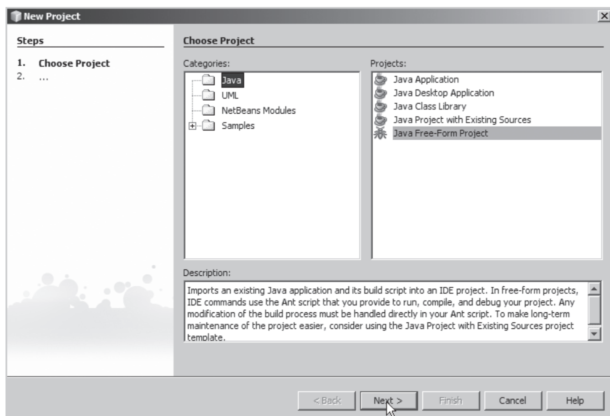


图　A-3

(4) 点击 Browse 按钮找到含有 build.xml 文件的 GameBook 文件夹，例如 C:/GameBook/server。NetBeans 会自动填写剩下的表单，你可以保留默认目录或者选择其他目录。点击 Next 按钮（图 A-4）。

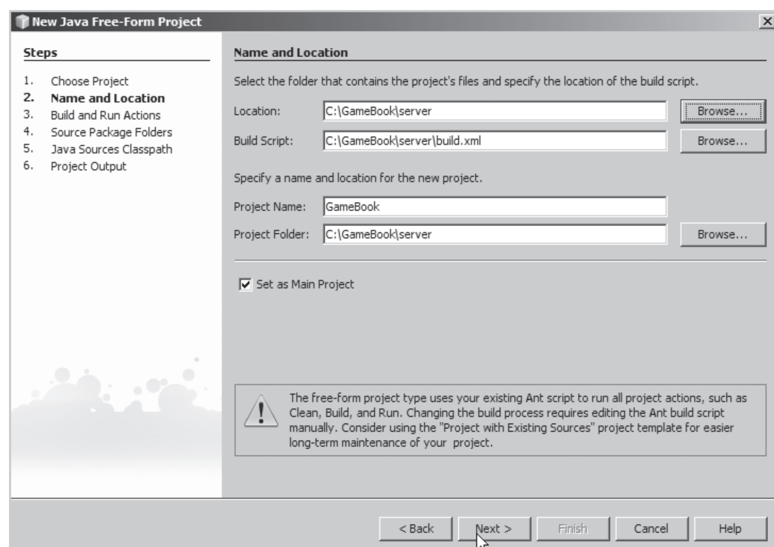


图　A-4

(5) 在下一个界面中，保留默认设置，然后点击 Next 按钮（图 A-5）。

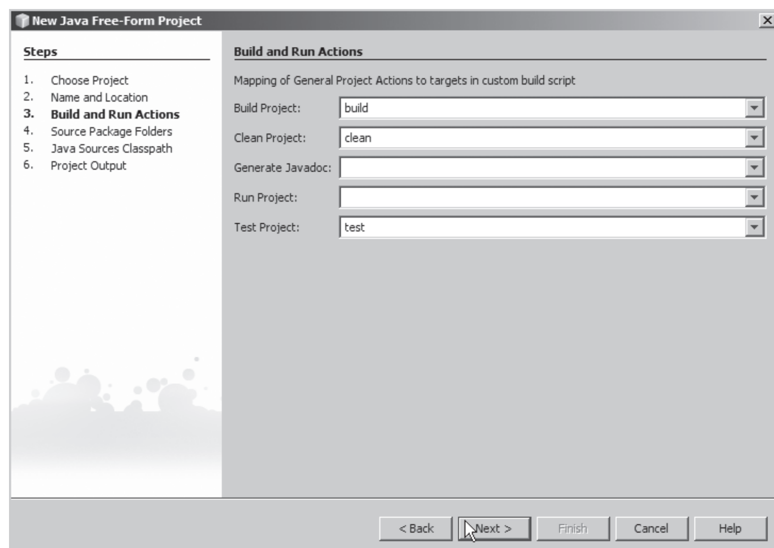


图　A-5

(6) 在 Source Package Folders（设置资源包文件夹）界面中，点击右上角的 Add Folder 按钮，然后定位到 C:/GameBook/server/src 目录（图 A-6）。

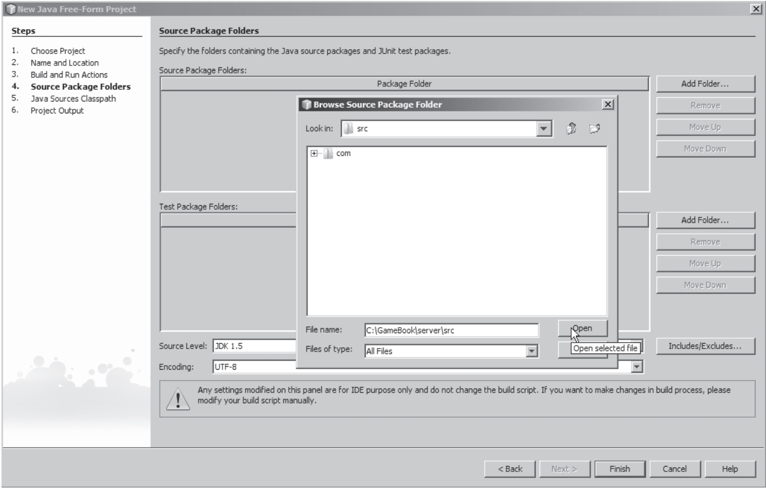


图　A-6

(7) 点击第二个 Add Folder 按钮，然后定位到 C:/GameBook/server/test 目录。之后你将看到如图 A-7 所示界面。

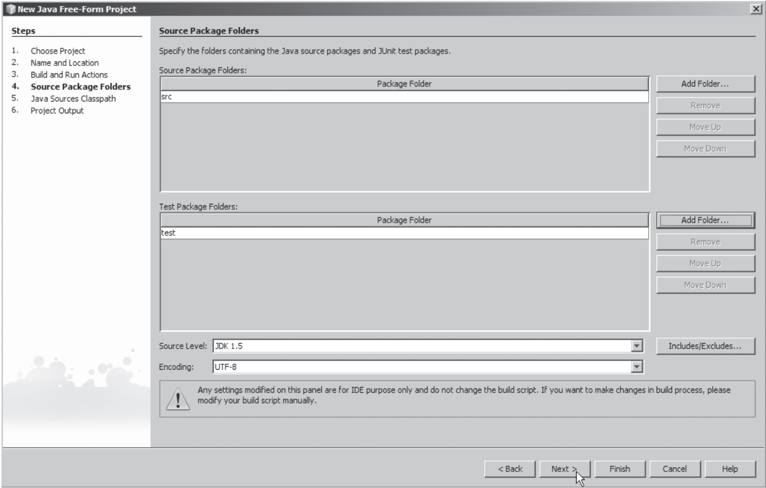


图　A-7

(8) 点击 Next 按钮进入 Java Sources Classpath（设置 Java 外部类路径）界面。点击 Add JAR/Folder 按钮，然后定位到 C:/GameBook/server/lib 目录（图 A-8）。

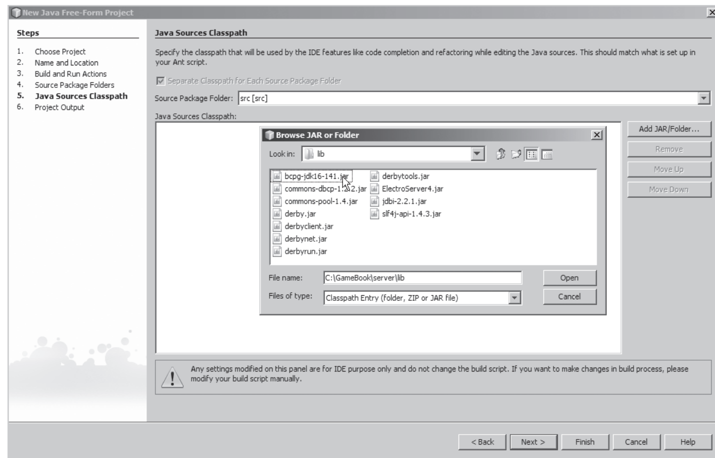


图　A-8

(9) 添加那个文件夹下所有的 .jars 文件。之后你将看到如图 A-9 所示界面。

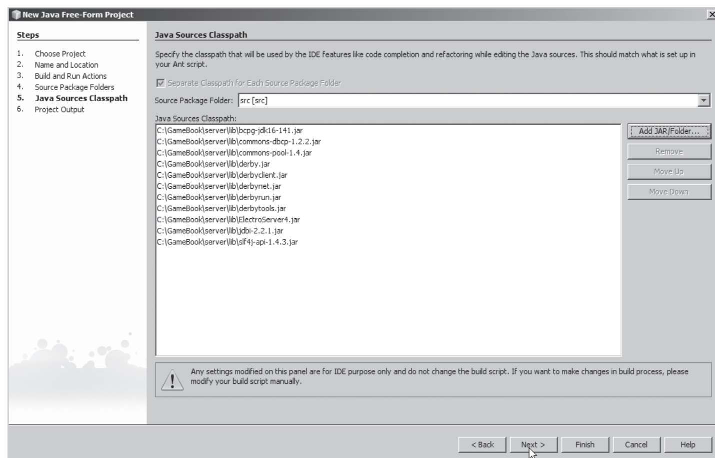


图　A-9

(10) 点击 Finish 按钮，NetBeans 项目就配置好了。如果想要编译的话，右击项目名（GameBook），然后选择 Build。

A.4.3　自动部署

自定义 ant 脚本被编译的同时也为你创建完整的扩展包。

(1) 找到 build.properties 文件。它应该和 build.xml 在同一文件夹下。

(2) 编辑 build.properties，使它指向本地的 ElectroServer4 的安装目录（或者是你手动复制放置扩展包的位置）。

(3) 在 NetBeans 中，右击 GameBook，然后选择 Test。

(4) 查看你在 build.properties 中设置的目录，看看扩展包是否奇迹般地出现了。

A.4.4 “古老家园”的自动部署

当你开始调试与“古老家园”相关的程序时，你将需要变更自动部署以及服务器级组件。

(1) 在 NetBeans 中，右击 GameBook，然后选择 Properties。

(2) 点击 Build And Run 按钮。

(3) 如果你没有看到名为 oldWorldTest 的 ant 目标，就点击 Add 按钮，在 Ant Target 下面的空白区域，选择 oldWorldTest。

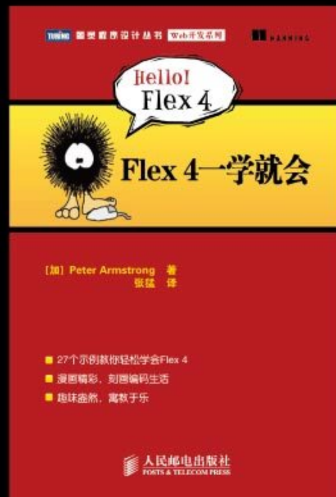
(4) 点击 OK 按钮。

(5) 右击 GameBook，然后选择 oldWorldTest。

(6) 查看你在 build.properties 中设置的目录，看看扩展包是否奇迹般地出现了。你可能会发现这个过程和前面的过程有所不同，这是因为这个在 GameBook 内有一个配置文件夹而之前的没有。

关于如何配置数据库和配置服务器级组件，请参考 A.2 节。

27个示例教你轻松学会Flex 4



丰富的经典示例和专有技巧



Adobe技术专家力作



ActionScript大型网页游戏开发

ActionScript for Multiplayer Games and Virtual Worlds Learn multi-user interaction concepts from the experts

亚马逊读者评论

“市面上关于多人游戏开发和虚拟世界构建的书并不多，这样一本重量级图书的出版无疑是给我们这些开发人员的一道福帖，即使是我这样的新手也从中收获良多。强烈推荐！”

“本书语言通俗易懂，结构严谨合理，是对当前网页游戏开发经验的总结。如果你想让自己的游戏开发水平更上一层楼，选择这本书绝对错不了。”

“Jobe Makar所领军的Electrotank是业内的顶级提供商，他所操刀的这本书也堪称网页游戏开发的实战宝典。书中不但有提纲挈领式的指导，还有大量详尽的实用技巧。我把它推荐给所有网页游戏开发人员，相信你们也一定会和我一样获益匪浅。”

目前对多人游戏和虚拟世界的需求呈现出爆炸式增长的势头，许多公司希望通过它来提高社交网站的黏性，广大的游戏开发者则对这一领域充满了激情。开发多人互动内容虽然具有一定的挑战性，但也不像想象的那么难，这是一段充满乐趣且回报丰厚的探索之旅。

本书阐述了多人网页游戏的许多基本概念，以及如何使用ActionScript将其实施到项目中。读完本书，你将掌握：

◇如何连接用户来实现实时交互；

◇何时选择在服务器或客户端进行游戏逻辑裁决；

◇时间同步技术；

◇通过航位推测平滑算法隐藏网络延时；

◇区块式游戏的等距视图技术；

◇对虚拟世界中的化身进行定制和渲染的技术。

利用这些知识，你也能开发出实时多人合作游戏，构造出自己的虚拟世界。

New Riders

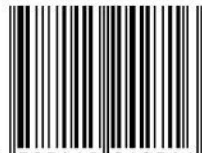
图灵网站：www.turingbook.com　热线：(010)51095186

反馈/投稿/推荐信箱：contact@turingbook.com

有奖勘误：debug@turingbook.com

分类建议　计算机/网页设计/Flash

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-24271-6

定价：45.00元